B.3 Test Server

On the test server a software image is running for each project. These images are built with the image building process and therefore contain the build software running in testing configuration. They poll in regular intervals for new testing configuration maps. If a new map is found, the map is loaded, all relevant tests (including SLint tests) are run, and finally a mail with the test results is sent to the responsible developers. (The set of packages to test and the developer mail addresses are stored in the configuration classes.)

C Coding Conventions

Most of the rules below originate in the book "Smalltalk Best Practice Patterns" from Kent Beck (a draft is available online `http://tinyurl.com/d7euby`). The page numbers these rules appear in the book are indicated. The book provides more information, examples, and the rational of these rules.

1. **Use type suggesting parameter names (p174)**
   Name parameters according to their most general expected class, preceded by "a" or "an". If there is more than one parameter with the same expected class, precede the class with a descriptive word or, if the class is Object, use just descriptive names.

   An Array that requires Integer keys names the parameters to at:put: as

   ```
   at: anInteger put: anObject
   ```

   A Dictionary, where the key can be any object, names the parameters:

   ```
   at: key put: value
   ```

2. **Put zero or one argument messages on the same line as their receiver (p176)**

```
foo isNil
a < b ifTrue: [ ... ]
```

   Exception: use the following formatting for boolean logic:

```
= aDefinition
   ↑ super = aDefinition
      and: [ self content = aDefinition content
         and: [ self link = aDefinition link
            and: [ self mail = aDefinition mail ] ] ]
```

3. **Put the keyword/argument pairs of messages with two or more keywords each on its own line, indented one tab (p176)**

```
a < b
   ifTrue: [ ... ]
   ifFalse: [ ... ]
```

Exception: if the message is really short, put it on one line:

```
array at: 5 put: #abc
```

4. **Start a block without a line break and end it on the same line as the last statement, and add a space between the brackets and the code**

   If the statement in the block is simple, the block can fit on one line:

```
a < b ifTrue: [ self recomputeAngle ]
```

   If the statement is compound, bring the code of the block onto its own line and indent:

```
a < b ifTrue: [
   self clearCaches.
   self recomputeAngle ]
```

   Close multiple blocks on one line.

5. **Use the statement separator, “.”, only to separate two statements**

   No dot before closing a block:

```
a < b ifTrue: [
   self clearCaches.
   self recomputeAngle ]
```

   No dot at the end of a method:

```
foo
   self clearCaches.
   ↑ self recomputeAngle
```

6. **Call the parameter ”each”. If you have nested enumeration blocks, instead use a descriptive word (p183)**

```
1 to: self width do: [ :x |
   1 to: self height do: [ :y | ... ] ]
```

7. **Use a Cascade to send several messages to the same receiver. Separate the messages with a semicolon. Put each message on its own line and indent one tab. Only use Cascades for messages with zero or one argument.**

   Instead of:

```
| parent |
parent := self listPane parent.
parent color: Color black.
parent height: 17.
parent width: 11.
```

   ...write:

```
self listPane parent
   color: Color black;
   height: 17;
   width: 11
```

   If (and only if) you need the value of a Cascade, append the message #yourself.

8. **Communicate important information that is not obvious from the code in a comment at the beginning of the method (p40).**

```
identifier
   "Answer the CSS identitfier of the receiver."
   ↑ self definition name
```

   Try to communicate information through code rather than comments. Information that needs comments for example are method dependencies (a method that needs to be invoked before another one). Furthermore, method comments to explain business rules can be helpful, preferably with a link to the feature task ID.

   Often, though, comments can be avoided. Instead of writing:

```
(self flags bitAnd: 2r1000) = 1 "am I visible?"
   ifTrue: [ ... ]
```

...implement a new method with an *intention revealing selector*:

```
isVisible
    ↑ (self flags bitAnd: 2r1000) = 1
```

...and the above code becomes

```
self isVisible ifTrue: [ ... ]
```

Avoid comments within the method body.

9. **Only use English for method names and comments.**

   Also find a consistent translation for the business domain. For instance, use powerDiscount for the term "VVG Start-Rabatt".

10. **Categorize all methods.**

    Use the following categories if appropriate:

```
accessing (direct accessors without indirect side effects)
accessing-dynamic (accessors with computation, e.g., derived values)
accessing-defaults (constants, creation of initial values)
accessing-descriptions (Magritte)
actions
rendering (entrypoints and top-level rendering methods)
rendering-... (helper rendering methods)
initializing
private (methods explicitly not part of the API)
```

11. **Commit code to the repository.**

    Commit often. Give branch or version number and a short list of feature/fixes. Say if the new code is "work in progress" or "experimental":

```
2.2.2

- RSS
- movie extension

work in progress
```