Lessons Learned with Eiffel 3: The K2 Project

Philippe Lahire & Jean-Marc Jugant
lahire@hibiscus.unice.fr, jmj@tigre.unice.fr
I3S – URA CNRS 1376 – University of Nice – Sophia Antipolis
250, avenue Albert Einstein, Sophia Antipolis, 06560 Valbonne, France

Abstract

K2 is a software designed for training people about agricultural and economical aspects that is currently developed by the team OCL at the I3S laboratory (University of Nice – Sophia Antipolis, France), for The Food and Agriculture Organization of the United Nations. Present state of the software deals with about one thousand classes, so that we thought it was a quite reasonable size for getting interesting feedbacks from the Eiffel language and ISE programming environment. Main topics that have been addressed are reusability and evolutivity of components and software maintainability.

1 Introduction

K2 is a new release of the Food and Agriculture Organization (FAO) of the United Nations Computerized system for Agricultural and Population Planning Assistance and training (CAPPA) software. It aims at simulating economic facts on a country scale to train people in economic analysis, so that it is possible to test an economic plan before applying it to the country. The applications are numerous:

- Simulations can be made to know when self-sufficiency could be reached: the user is able to modify data, such as imports or production priorities (i.e. the user decides to focus the production on particular products), and see what happens.
- The software helps the user to establish environmental assessments; for example, one can see the impact of deforestation on natural resources, or one can simulate the effects of an excessive dryness.
- Demography is taken into account in its corresponding module: the population growth rate is estimated and it is used in other modules, e.g. to assess needs in food or to evaluate urban migration.

History

CAPPA first came up in 1980 and has been used as an economic analysis tool, whereas its purpose was merely educational. It was developed with tools available at that period: the basic (the language!) program had to fit on a floppy disk and was designed for alphanumeric terminal. Since 1980, new technologies have been developed, therefore new specification requirements arose:

- more flexible data structures,
- creation and use of a genuine database,
- a toolkit instead of a monolithic program
- a graphical user interface (GUI),
- data importation/exportation from/to other databases or spreadsheets.
- more powerful economical analysis

The latest version of CAPPA was too monolithic and platform oriented (optimized code and space for PC) to offer any support for software re-engineering, and the specifications evolved a lot. K2 had then to start from scratch

Environment

After a rough analysis of the project, the language to use should be able to deal with general concepts and GUIs. The numerical aspect of the software was not to be considered since the calculations did not require a high level of precision neither

1Interactive Software Engineering, Santa Barbara, California.
complex operations. The object oriented programming was chosen to avoid CAPPA’s drawbacks mentioned earlier, and to provide nice engineering features:

- modularity and encapsulation,
- easy adaptation to specifications requirements evolution,
- reusability

K2 foreseeable size is around 1500 classes (without library classes) [Rousseau 91].

According to such requirements, we focused on two programming languages:

- C++ [Stroustrup 91], widely known, comes with non-homogeneous libraries, and its use is not straightforward [Joyner 92], though the language has evolved since we have started the project\(^2\). C++ relies on C and inherits its drawbacks.

- Eiffel 2.3 [Meyer 91], at that time more confidential, is especially designed for software engineering: assertion mechanism, powerful multiple inheritance, constrained genericity, easy readiness, prototyping ability.

Because of all the requirements of our application, its size and the features of the programming languages, we decided to use Eiffel instead of C++.

**Development constraints:**

- Non homogeneous requirements specifications: several consultants are responsible of the different K2 modules and use their own formalism to specify them.

- Time-randomized specifications: because of the organization, we did not get all the specifications at the beginning of the project. We were not able to design the whole project at once; this incrementality leads to rewrite some components already set.

- Two people are programming the application. In order to handle the different version of the components, we use the configuration manager CVS [Cederqvist 93].

**Platform constraints:** the specification was to provide this software on a PC system with Windows 3.1. As Linux becomes popular, it would be interesting to release K2 on a such platform.

\(^2\)It incorporates now templates and exceptions.

## 2 Development: Issues and Results

We started the development with Eiffel 2.3 on Unix workstations, because it was not available on PC.

Now only Eiffel 3 is distributed on PC: so we had to port the application from Eiffel 2.3 to Eiffel 3.

The graphics library (Eiffelvision) changed a lot between Eiffel 2.3 and Eiffel 3

According to these facts, we discuss about portability and statistics on the development aspects of K2.

### Portability

**Eiffel 2.3 to Eiffel 3** At the language level, migrating from Eiffel 2.3 to 3 was mainly made by using ISE’s translator, allowing old compiled classes to be upgraded to their Eiffel 3.0 counterparts. Mostly, the generated classes fit our application, i.e. the new components matched the old requirements. In fact, the awkward part - where the translator failed - was due to some complex inheritance graphs and to the new Eiffel 3 library design.

The use of these libraries makes us change our application. This costs a lot because of the size of K2. In particular, the class environment, which deals with persistent objects, lacked [Lahire 92]. Fortunately, ISE should provide soon a library (EiffelStore) to handle persistency in a more transparent way (but not as much as if it were included in the language [Chignoli 94]) and to communicate with databases. The graphic library also evolves a lot, proposing now a common interface with Motif, Openlook and Windows in the near future: so we had to entirely rewrite the graphic interface of K2 in order to fit the new graphics library.

### Unix to PC/Windows

When the project should have been ported, the only Eiffel 3 compiler on PC was SiG’s Eiffel/S 1 21. We then faced some difficulties:

- Libraries were not compatible. We tried to build a layer on the Sig’s components in order to get the same interface as the ISE’s components. The design was quite simple, though ISE’s libraries are richer than SiG’s.

- Compilers accepted a different syntax. We found out that Eiffel/S was more compliant to the definition of Eiffel 3 than ISE’s first release, despite it did not accept the keyword old and the rescue, strip statements.

- The PC environment for Eiffel/S was poor, with just a command-line compiler.
• Eiffel/S did not provide graphic libraries. This was a huge drawback because K2 must provide a GUI. This last point makes us aware not to use Eiffel/S but to find another solution which includes graphics handling.

Graphics  To sort out the problem of support in graphics, some solutions have been evaluated:

• Re-writing the entire EiffelVision library: implementing the abstract layer of Eiffelvision using the Windows 3.1 API. The cost of this task would have been around height months and we found it to risky.

• Switching the development to C++ because some of the C++ distributors provide also a graphic toolkit. This means to use the Eiffel implementation of K2 as a conceptual scheme of the targeted C++ implementation. This sounds tricky because of the lack of constrained genericity and multiple inheritance in C++, and the impedance mismatch between these two languages.

• Developing external modules and maintaining them in parallel with our application. This means to move the non-graphic components of K2 to the PC+Eiffel/S, and then to interface it with some ad-hoc C routines. The maintenance and even the development (and resulting debugging) would become very cumbersome.

Final Decision  At that level of the development, C++ could have been an acceptable solution according to graphic interface. But ISE announced the Personal Eiffel for Windows (PEfW). So we decided to rely on this company to port its graphic library (this solution ensures a full compatibility with the Unix libraries).

K2 figures

Currently, the repartition of the development is described in figure 1. About the data structures of K2, one can notice that the main part is related to graphics, though the total time spent on callbacks (implementing commands associated to widgets), can be considered far smaller than the time spent for application specific components (a.k.a. graphic behaviors in figure 1); this is the reason why productivity results are also given in figure 2 without callbacks.

The reuse level of the libraries is high: half of K2 relies on them. From another point of view the productivity gained a factor 2. Whereas, this productivity makes sense for an one man.month effort, the generalization to several persons is not straightforward, due to the amount of time needed to develop

<table>
<thead>
<tr>
<th>Domain</th>
<th>% in K2</th>
<th>Class count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Callbacks</td>
<td>39.5</td>
<td>206</td>
</tr>
<tr>
<td>Reusable Structures</td>
<td>21.5</td>
<td>112</td>
</tr>
<tr>
<td>Application Specific</td>
<td>13.3</td>
<td>69</td>
</tr>
<tr>
<td>Graphic Descriptions</td>
<td>8.0</td>
<td>41</td>
</tr>
<tr>
<td>Misc. Graphics</td>
<td>7.3</td>
<td>38</td>
</tr>
<tr>
<td>Graphic Behaviors</td>
<td>6.5</td>
<td>34</td>
</tr>
<tr>
<td>Handling Persistency</td>
<td>3.9</td>
<td>20</td>
</tr>
<tr>
<td>Sub-total</td>
<td>100.0</td>
<td>520</td>
</tr>
<tr>
<td>EiffelBase &amp; EiffelVision</td>
<td>447</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>967</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: Data Structures in K2

the core of K2. Indeed parts of the reusable and application specific structures, and parts of the database implementation were prerequisites for any team development.

The documentation and design efforts were included in the evaluation presented in figure 2, so that the productivity in terms of coding is higher. From now, the productivity is likely to decrease because of two factors:

• The size of K2-specific classes is greater than the library size

• No more distributed libraries are available for K2.

According to figure 3, 87% of the K2-specific classes have less than 150 lines of code, and 50% of the K2-specific classes have less than 50 lines of code. This big proportion is mainly due to callback encapsulations. Almost no extra C has to be written except a small layer to interface with the operating system, whereby the maintenance would focus on Eiffel.

K2 being at an early stage of development, more information will be available and statistics will be more reliable and could be compared to the study in [Coulange 93].

3 ISE Eiffel 3: Lessons Learned

This environment provides many facilities through Ebench, even if some of them can be improved. Following critics are related to the release 3.2.3b of the environment

Library and Run-time Issues

EiffelBase [Meyer 94] supports the data structures we needed, especially the tree structures. The support for persistency corresponds only to one class, the class storable which is not sufficient for implementing the management of persistent objects in a transparent way.
### Aspects of K2

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C Externals</td>
<td>3 functions (to access Unix environment variables and time)</td>
</tr>
</tbody>
</table>
| Lines of Code (LOC) (per K2 class) | Minimum: 12  
Maximum: 1,458  
Average: 106 |
| Total                         | 55,282                                                                   |
| Size of the executable        | 14.67 Mo  
Stripped: 11.38 Mo  
Optimized (except library code): 6.83 Mo |
| Productivity                  | Average: \( \approx 27 \) classes/month  
Without coding callbacks: \( \approx 16 \) classes/month  
Including reuse: \( \approx 51 \) classes/month  
Reuse & without callbacks: \( \approx 30 \) classes/month |

Figure 2: Project Statistics

![Distribution of LOC/K2-class](image)

Figure 3: Distribution of LOC/K2-class
The class **storable** does not provide databases features such as incremental loading of data or independence between the physical storage of objects and the conceptual layer. We had to implement those facilities in classes, for instance the class **simple.repository** or the class **complex.repository** over the class **storable**.

**Documentation**

Documentation is automatically produced by a tool called **short**, which generates the interface of a given class, and by a tool called **flat**, which generates the set of routines of a class, including those which are inherited. A combination of the two (a **flatshort** view) is also possible. Other facilities are also available such as the list of clients, suppliers, ancestors and descendants of a class, and the list of implementers of one feature.

Though it is possible to save those views into a file, a troff\(^3\) output through **Ebench** would have been useful. It would have been also interesting to get a tool creating automatically the documentation of a set of classes.

For reverse engineering, **Ebench** should be able to draw – or produce an equivalent document of – the inheritance graph of a system (as the tool **good** in Eiffel 2.3). Though this functionality is present in **EiffelCase** [Nerson 92], a minimal clever tool should be provided, in order to be able to manipulate the abstract syntax tree produced by the compiler. A query mechanism could be associated with class indexes, e.g. to produce a documentation related to a given topic: for example one might want to edit classes with the field **version** and with the version number greater than 3.2.

**Melting Ice Technology**

This is a very nice feature provided by the environment; the development time which is saved is consequent, except when **Ebench** crashes at pass 4, while it is freezing the system of classes: the whole system should be more robust. Our system freezes on a per-night basis (using a batch command) or on a per-week basis, avoiding the operating system to be overloaded; the impact on our application efficiency was not relevant when interpreting melted code.

**Editing**

The text windows provided as editing supports offer a minimal set of functionalities (cut and paste, \ldots). A true syntactic editor would have been nice, though it is possible to use our own canvas-driven editor. So far, a single-click browser would improve the speed (avoiding the drag-and-drop operations), i.e. one click on a class name makes it appear in a window (**html-like**).

**Debugging**

Most of the time, debugging was needed because of a logical error in our application and not because of memory leaks, for instance.

When debugging an application, some operations are not yet available:

- to put breakpoints in the body of a feature,
- to execute code step-by-step,
- to trace variables and to edit them interactively through **Ebench**,
- to switch on, at cluster level, a particular trace (i.e. a labeled debug statement).

The Eiffel assertion and exception mechanism were helpful to locate the possible source of program crashes.

**Cluster Management**

The LACE\(^4\) language allows to describe the system to be **built** by the compiler. The clusters used by LACE map the **Unix** directories where the classes are; it handles cluster level and class level options, such as debugging, exportations or renamings. Since the SDF used for Eiffel 2.3, an evolution took place according to cluster management, but the descriptions remain at a module level, i.e. there is no object model of cluster management, hence no meta-manipulation is possible.

**Constants**

A typical application has constants:

- to parameterize the behavior of the application,
- to ensure consistency of messages (errors, warnings, \ldots) and labels (window titles, buttons, \ldots),
- to implement a default set-up (directories, user preferences, \ldots),
- to be language independent i.e. to be able to deliver the software for English, French and Spanish people.

\(^3\)or **Ltex**, encapsulated **Postscript**, \ldots

\(^4\)Not part of the Eiffel 3 definition.
At first, the constants were put in a few classes at the top of our class hierarchy so that they were available to everyone. But at development stage, when modifications often occur, the melting took too much time. So, the best way of implementing constants is to spread them across several classes, which are inherited only by the relevant classes. The constants used for Motif are described in the Motif resources. Now the addition of constants requires only about 15% of the application, to be melted again.

4 Object-Orientation and Reuse: Lessons Learned

What kind of Reuse?

Any application, designed with care, is likely to reuse components and to yield new reusable components. This well-known statement must be corrected to take into account two different kinds of reuse:

- A given application may reuse any other application domain (AD), e.g. softwares dealing with rental administration systems.

- A given application may reuse any general domain\(^5\) (GD), e.g. sorting algorithms, linked lists.

The reuse cases as shown in figure 4, where \(A_0\) stands for the last developed application and \(A_n\) for an application to be developed, will be discussed.

<table>
<thead>
<tr>
<th>(A_0)</th>
<th>Case 1</th>
<th>Case 2</th>
<th>Case 3</th>
<th>Case 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GD</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Figure 4: Possibilities of Reuse

Case 1 The new application \(A_n\) is close to \(A_0\). Obviously, this is the best configuration to obtain a good level of reuse. This case happens:

- When \(A_n\) is \(A_0\) with a few (or no) change(s) in the requirements: the activity involved is adaptation, i.e. inheritance with a few change on classes. If \(A_n\) is a new release of the product \(A_0\), then it is adaptative maintenance.

- When \(A_n\) belongs to the same problem domain from \(A_0\). It is likely that entire frameworks would be reused.

This can also happen when \(A_0\) has been split into logical units (modules in \(K2\) are tools using general \(K2\) data structures and some private data): each module reuses the common data structures.

Case 2 \(A_n\) uses the same general structures as \(A_0\) does. This seems to be the general case: an application relies on a development environment, e.g. a compiler and its libraries. Hence the choice of the latter may affect the quality of reuse and the robustness/reliability of the former. Most of the commercial libraries come with low-level objects, e.g. basic types such as strings or integers, generic lists.

Case 3 \(A_n\) and \(A_0\) belong to the same application domain. Typically, this denotes a shift to a new user-transparent technology, e.g. the application becomes distributed.

Case 4 \(A_n\) was developed from scratch. The development effort has been made on both domains: library and application.

Reuse beyond the Limits?

In the previous section, we spoke about the main reuse levels. Now we would like to give examples of some situations showing reuse limits in project such as \(K2\).

We discuss language reusability limits according to multiple and repeated inheritance, genericity and persistence capabilities. In most cases, these mechanisms allow to reach a satisfactory level of reuse, but sometimes they lead to the opposite effect according to readability, maintainability and efficiency. In such case, components are reused but not appreciated [Rousseau 94].

Reuse and Maintainability: Persistency

\(K2\) makes projection of economic data through time. This data (aggregates) are structured according to nomenclatures, one nomenclature being a tree with typed nodes, it is another way to say that an aggregate is a tuple of nodes.

If produced quantity depends on the scenarios that may be addressed, nomenclatures are not dependent of hypothesis that may be made within scenarios. So that several solutions may be chosen by Eiffel programmer that only gets the class storable:

- to create only one file that contains both data from scenario and all nomenclatures:

- to create several files that contains either one nomenclature or a set of data, linked to one scenario

Solution putting together nomenclatures and scenario data is not realistic because it would lead to data inconsistency if...
nomenclatures are not changed within all scenarios.

From our experience, data associated to scenarios need at least 10 to 12 Mo, and the final user will handle several dozen of scenarios, so that the solution to handle only one file cannot be retained. But the solution that we chose had some drawbacks such as the need to implement the attachment of nomenclature to scenarios through functions that find files according to a minimal set of information.

In another part, the class **storable** is in theory reusable but practical examples show that, for applications handling quite a large amount of shared data, the use of this class is too much inefficient, even if the addition of components that handle a memory buffer improves performances.

**Proposal:** It sounds important that Eiffel language provides persistency mechanisms allowing access to existing databases but also the incremental management of persistent object loading, freeing and updating, and the support of selective queries. This means to provide new library components and a new run-time [Lahire 92].

**Reusability and Evolutivity: repeated Inheritance**

To implement some entities using repeated inheritance instead of client/supplier relationships shows some limits in the practical use of repeated inheritance. Reading of class source code becomes tedious even while class is in the development phase, because of the complexity of the **redefine, undefine, select, rename** statements.

The implementation of a hierarchy of classes implementing facilities associated to different kind of Motif displays (with or without scrolling, managing data following one, two or three nomenclatures according to user choices, providing also generic facilities for modifying data shown through a display) makes us aware of this problem.

Data mentioned above are organized in such a way that it is possible to access them at different aggregation levels. For instance, one may consider the production of cereals or the production of wheat or the production of a specific range of wheat, in one rural area (noted “R”) of the country.

To display data following a nomenclature (e.g. the nomenclature of products) means that if we consider the above example, the quantity produced in “R” of all products will be put on screen.

The implementation relies on several hierarchies of classes:

- definition of graphical cells associated to display (**graph.cells** and its descendants),
- management of iterations on cells in order to handle exchanges between the display and the persistent structures where the data are stored (**graph.cell_iterator** and its descendants),
- queries which implement the contents of exchanges (**graph.cell.query** and its descendants).

In present case the hierarchy is complex and raises several comments. On the first hand, they will to reuse classes such as **graph.cell.query**, **graph.cell_iterator** and **graph.cells** in order to implement the classes **graph.cell.flex.query**, **graph.cell_iterator3** and **graph.cells3** leads to classes with complex inheritance statements: they are a combination of **undefine, select, redefine, rename** which are interdependents, as shown in figure 4; this reduces the readability of these classes. On the other hand, it makes possible not to duplicate common source code, but does not facilitate class maintainability so that it reduces its evolutivity.

```plaintext
class CELLS inherit

GRAPH CELLS
rename
make as make.2d,
generate as generate.2d,
item as item.2d,
put as put.2d, ...
undefine
item.2d, put.2d,
line.execute, column.execute,
line.item, column.item, execute, ...
redefine
attached, fields,
display_query, update_query, ...
end

GRAPH CELLS
rename
make as make.2d,
item as item.2d,
put as put.2d, ...
undefine
item.2d, put.2d,
line.execute, column.execute,
line.item, column.item, execute, ...
redefine
attached, fields,
display_query, update_query,
generate, ...
select
generate, ...
end

GRAPH CELL ITERATOR3
```
rename
   item as gc_item,
   put as gc_put, ...
undef
   copy, consistent, is_equal, setup
redefine
   attached
select
   gc_item, gc_put, ...
end

K2.ARRAY3 [GEN_GRAPH_CELL]
rename
   make_2d as array2.make,
   make as syn.array2.make,
   make_3d as array3.make_3d
end

-- Etc...
end -- class CELLS

class HEIR inherit

PARENT
   redefine
   f using old_f
end

feature
   -- Etc...
end -- class HEIR

Figure 5: Example of view description

Figure 4: Example of one complex inheritance statement of K3

Proposal: Most cases of repeated inheritance deal with a need to redefine a routine and to re-use in the new body the old one. In release 2.3 of Eiffel the writing of inheritance statement was less orthogonal and less self-explanatory but more concise. It could be interesting to keep the syntax of Eiffel 3 but to introduce facilities making possible to build views of the Eiffel 3 source code. One of this view could be described as in figure 5 and should be more readable than only with the Eiffel syntax (figure 6).

Genericity and Reuse

We briefly consider the case of class-genericity. ISE compiler implementation limits to four the number of generic parameters; potentially, this restriction reduces class complexity which may remain too high when programmer uses several level of genericity, e.g. linked_list [tree [persistent.data]]. Such a thing leads to readability problems induced by the multiplicity of embedded routines with generally the same name, allowing to cover different levels of genericity. This readability problem is very difficult in the development and testing phase, but also in the documentation where people have to switch from one class to the other in order to understand the meaning of operations.

Figure 6: Usual inheritance statement in Eiffel?
Proposal: It could be interesting to provide tools for exploring different levels of genericity making possible to see quickly the routine corresponding to one genericity level.

5 Conclusion

K2, the FAO’s Computerized system for agricultural and population planning assistance and training software, is a constantly evolving software, due to specifications evolution; to cope with the changes is achieved by using object-oriented techniques, largely supported by Eiffel 3, and trying to design the best reusable components. The statistics clearly show that the project has come to a new phase in its development because it started reusing non-vendor library classes.

From our experience, we think that when people build large software, whatever are the skills of the language used, the view of a class through the syntax only is not sufficient. The ISE environment provides already a number of features which favour the reuse of class libraries but we think that it should be possible to do even better. In particular, it should be interesting to get tools displaying on screen the information related to the programming action. Information displayed by those tools could not follow exactly the syntax of the language but use an appropriate syntax (graphics, textual, . . . ) to the problem to solve.

With the help of ISE’s environment and tools and with new module specifications, we will be able to enhance our analysis of reuse and productivity, and we will propose an object-oriented case tool MOP, suitable for Eiffel 3.

Acknowledgements

We would like to thank Carine Fédèle and Philippe Collet for their help in the providing of the camera-ready copy of the paper.

References

[Cederqvist 93] CEDERQVIST P. Version Management with CV$S$, 26 Oct. 1993, 110 p. release 0.8, for CV$S$ 1.3-s2/1.3.1.


