M2K
An Approach for an Object-oriented Model of C Applications

Ignacio Cassol[1] and Gabriela Arévalo[2]
[1]*Facultad de Ingeniería, Universidad Austral, Buenos Aires, Argentina*
[2]*DCyT (UNQ), CAETI (UAI), CONICET, Buenos Aires, Argentina*
icassol@austral.edu.ar, garevalo@unq.edu.ar

Keywords: Reverse Engineering, Legacy Software, Procedural Language, Object-oriented Paradigm, Reengineering, Refactoring, Design Recovery.

Abstract: When analyzing legacy code, generating a high-level model of an application helps the developers understand how the application was structured and how the dependencies relate the different software entities. Based on useful properties that the object-oriented paradigm (and their supporting analysis tools) provide, such as UML models, we propose M2K as a methodology (supported by our own tool) that generates a high-level model from legacy C code and proposes differents refactorings. To understand how procedural-based applications were implemented is not a new problem in software reengineering, however our contribution is based on building automatically an object-oriented model and help the experts to define manually different refactorings that let the developer to improve the application. Besides a methodology and the supporting tool, we provide a summary of thirteen case studies based on small-scaled real projects implemented in C and we showed how the results validate our proposal.

1 INTRODUCTION

In the 70's and 80's, most systems were written specially in procedural-based languages, such as Cobol, C, Fortran, Pascal, and Clipper. Many of these applications are still working, but they are already legacy code. Their main problems are that mostly they were implemented in an ad-hoc way or, even without following an analysis- or design-based methodology. Usually, provided documentation is not enough, and stakeholders and original developers have left the software development group taking away implicit design knowledge that is not coded in the application itself. Due to all these problems, their structure and functionality are difficult to deal with during any maintenance task (van Gurp and Bosch, 2002). Though the solution for all these problems could be the replacement or the migration of the complete systems, this action is difficult because they are working currently in the companies and they represent their actual economic capital.

During last decades, object-oriented paradigm has been useful not only in implementing business applications, but also to implement analysis and refactoring tools. In this paper we propose a methodology to map procedural code into a high-level model, specifically a class model. By analyzing the source code, we identify a list of potential class definitions inferred from the code. Then, we let an expert to analyze them in order to identify class candidates and propose refactorings to improve the final model. The mapping from source code to class definitions is a automatic process, however, the further analysis to apply refactorings is done by experts manually.

The main reason to use an object-oriented design model is that by using mainly encapsulation, we limit the complexity of maintenance, and any proposed change can be applied in the model and in the code without affecting other entities.

We validate our methodology on thirteen case studies, but due to space limitations we present only one in detail and general information about the other case studies. The main goal of the proposal is to get a high-level model of C applications and offer improvements (through refactorings) at the design level. Then, the expert can use this improved model to implement a refactored application or modify the original one. Our work does not include this last step.

This article is structured as follows: Section 1 presented the context and the problem that our approach solves, Section 2 summarizes briefly the related work to our approach, Section 3 details the M2K methodol-

ogy to be applied in C applications, Section 4 shows in detail the application of M2K to one specific case study, Section 5 analyzes the applications of refactorings in the thirteen case studies, and finally Section 6 concludes our paper.

2 RELATED WORK

The related work will be summarized briefly in two subjects: source code analysis of C code to obtain a high-level model and model refactorings. From the source code analysis viewpoint, works related to identifying objects starting from C code are (G. Canfora and M.Munro, 1996), (Yeh et al., 1995) and (Siff and Reps, 1999) that propose different approaches to find classes in legacy code by identifying functions, variables and modules in the code, and to cluster these together based on different classification criteria. Fanta *et. al.* (Fanta and Rajlich, 1999) describe an approach that allows to restructure C code into a new C++ classes. Although the migration of the paradigm is relevant, compared to our approach the proposal does not count with a model. The closest one to our proposal is the work of *concept formation* (Sahraoui et al., 1997). This method relies on the automatic formation of concepts based on the obtained information directly from the source code to identify objects. Zou and Kontogiannis framework (Zou and Kontogiannis, 2001) proposes an iterative and incremental migration process. Our proposal differs from this one because M2K is not intended to migrate the code. Garrido *et. al.* (Garrido and Johnson, 2003) refactor C programs with the presence of condicional compilation directives. The approach is automatic and the refactorings offered are focused on the code. Compared to our approach, this approach does not offer a high-level model and is not dealing with changing from a paradigm to another one in a pure way. Another closest proposal is the use of language-indepent platform Moose (Nierstrasz et al., 2005) to analyze source code. In this latter approach, using different plugins, they are able to generate a model of the source code. This model is a representation of the C code in the FAMIX model, but they do not offer a migration to another paradigm. Recent works summarized by Yada *et. al.* (Yadav et al., 2014) are focused on generating visualizations of C applications, that could be considered as high-level models.

From model refactorings viewpoint, mainly UML models are considered as suitable candidates for model refactoring (Astels, 2002)(Boger et al., 2003) (G. Sunyé et al., 2001). Many of the refactorings know from object-oriented programming (Fowler et al., 1999) can be ported to UML class diagram as well.

From our knowledge, and based on the state of the art, there is no methodology that combines to generate a high-level model starting from C applications and propose improvements of the model using refactorings *a la Fowler*. We consider that the methodology we design to identify class candidates and the proposed refactorings are the main contributions of this paper.

3 OUR APPROACH: M2K METHODOLOGY

The M2K (the acronym of *Mapping to Know* or *Methodology to Know*) methodology analyzes C code that should fullfil the following preconditions: (1) The source code should not have syntactical errors, thus it can be compiled correctly without breaking down the system from the syntactical viewpoint; (2) the code should be written in Ansi C. We chose this target language because it is the most used programming language in industry [1]; (3) The system should use Abstract Data Types (ADT) to define the data structures.

3.1 Phases of M2K

M2K has two phases: *Source code Analysis* (supported by our tool *ModelMapper*) and *Expert mapping*. Just to clarify in the rest of the paper, we define the *model entities* as the meta-entities that are used to build a model, such as classes and methods, and *domain concepts* as the concepts that are implemented in a specific domain, such a customer in a business application.

The *Source Code Analysis* is automatic and is composed of two steps performing a static analysis.

1. **Extracting.** The tool extracts C model entities (variables, modules, ADTs and functions) from the source code by using a customized parser, and
2. **Mapping.** Using some heuristics, the tool maps the extracted entities to infer possible classes candidates (CCD). Specifically, ADTs are used to build potential classes, the functions are mapped as methods, variables as attributes and modules as classes.

The *Expert Mapping* phase is non-automatic and requires an expert that analyzes the group of CCDs in order to decide the final set of classes represented

[1] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

in the legacy code. The choice of final classes is based on the user knowledge, the functionalities of the legacy program and the refactorings proposed to improve the original model. In this step, concepts related to object-oriented paradigm appear in the model, such as associations or inheritance.

3.1.1 Source Code Analysis: Extracting Step

To understand how we analyze the source code, we illustrate the process using a simple C program, that has three modules that implement three data structures: a stack, a queue and a list of integers. Following we show the example code:

```
#include "list.h"
#include "queue.h"
#define TAM 6
#define MAX TAM-1

typedef struct {
   int top;
   int item[TAM];
}stack;

/*stack functions*/
int full(stack *);/
int empty(stack *);
void push(stack *, int);
void pop(stack *,int *);

void sortQueues(queue[]);

void main() {. . .}
```

The header of the `list.h` module is:

```
#define TAM3 6
#define MAX3 TAM3-1

int itemlist[TAM3];
int actual;

void initlist();
void insertafter();
void insertbefore();
void length();
void destroy();
int getitem();
int listfull();
int listempty();
void moveactual(int);
```

The header of the `queue.h` module is:

```
#define TAM2 6
#define MAX2 TAM2-1

typedef struct {
   int front;
   int rear;
   int length;
   int item2[TAM2];
   int cant;
}queue;

/*queue functions*/
void enqueue(queue *, int);
void dequeue(queue *,int *);
int isFull(queue *);
int isEmpty(queue *);
```

In the **Extracting** step, we parse and analyze the code in order to generate separate lists containing entities that represent global variables, functions, ADTs and modules identified in the legacy code. Thus, in our example the final result will be:

- $ADTs = \{stack, queue\}$
- $Functions = \{main, full, empty, push, pop, enqueue, dequeue, isFull, isEmpty, initlist, insertafter, insertbefore, length, destroy, getitem, listfull, listempty, moveactual, sortQueues\}$
- $Variables = \{TAM, MAX, TAM2, MAX2, TAM3, MAX3\}$
- $Modules = \{list, queue\}$

The ADTs are extracted based on the `struct` statement, the functions are extracted based on the function definitions and modules are extracted based on the `#include` sentence in the header of each file in the application. In the specific case of global variables, we extracted them from `#define` in the header, and variables defined in the header of the file that contains the `main()` function.

3.1.2 Source Code Analysis: Mapping Step

In the **Mapping** step, we look for relationships and dependencies between the lists generated in the *Extracting* step and create the CCDs. To perform this phase, we have defined a set of heuristics. Each one is defined with a set of conditions to be fullfilled by model entities, and the steps to create a new CCD, or modify an existing one based on those entities. We define heuristics because they serve as checks or indicators by which a structure may be examined for potential improvements (Yourdon and Constantine, 1979). The heuristics rules may result in non-unique assignment, e.g., a function with two ADT parameters may be assigned to both resulting CCD. These conflicts are recognized and resolved manually in the *Expert mapping* phase.

ADT-based Heuristic. Based on the fact that each ADT in C code is defined using `struct` and that groups a set of typed variables $\{a_1, \ldots, a_n\}$, with this heuristic we look for cohesive entities by creating a CCD with the same name as the ADT and the following pair of elements $(\{a_1, \ldots, a_n\}, \{f_1, \ldots, f_m\})$, where each f_i is a function that takes an entity of the type ADT as arguments or return it. In our example, the CCD *stack* is created as $stack = (\{top, item[TAM]\}, \{full, empty, push, pop\})$, and the CCD *queue* is created as $queue = (\{front, rear, length, item2[TAM2], cant\}, \{enqueue, dequeue, isFull, isEmpty\})$

In the case that the argument is a set of elements typed with the ADT, we do not keep these functions in the defined CCD, because those arguments can build

a new one that deals with it. In our example, the function `sortQueues` is not included in the previously defined CCDs, because the parameter is an array of queues.

Module-based Heuristic. From design viewpoint, the modules define (and implement) functions that share a common goal or are related by a common functionality, in our example the module `List` defines several functions to deal with list operations. However, this design principle is not always fulfilled in all C applications. To our analysis, we consider that the modules are implemented in a `.c` and/or `.h` files. In this heuristic, depending on the structure of the module, there are three possibilities of encoding it in our model.

1. If the module contains only a set of functions that are not related to a CCD previously defined, we will generate a CCD with those functions mapped as methods and with attributes, if there are declared variables. If not, the defined CCD will have only methods.

   In our example, this heuristic will create the CCD *list* as $list = (\{itemlist[], actual\}, \{initlist, insertafter, insertbefore, length, destroy, getitem, listfull, listempty, moveactual\})$

2. If there are only ADTs, and all the functions are related to those ADTs, we will get one CCD per each ADT according to the *ADT-based heuristics*. In this case, no new CCD will be created.

   In our example, the module `queue.h` is not building a new CCD because it was already mapped as the CCDs `queue` in the ADT-based heuristic.

3. If there is a mixture of declaration of variables and ADTs, and/or functions related or not to those ADTs, we will generate CDDs for each ADT with their corresponding methods - according to the *ADT-based heuristics* -, and another CCD with the same name as the corresponding module with the declaration of variables and the corresponding methods not related to a CCD.

   In our example, a new CCD named `main` is created containing the method `sortQueues` that was not related to any ADT because its argument is an array, and is not considered in the ADT-based heuristic. The rest of the functions are related to the ADT `stack` and are defined in the corresponding CCD.

Global Variables-based Heuristic. In this heuristic, we analyze each global variable y and a function $f(x_1,\ldots,x_n)$ that reads or writes y in its definition, and then the CCD that contains f adds y as a new attribute in its definition. Thus, this heuristic does not create CCDs. With this heuristic, we identify a cohesion-based situation that helps us to complement an existing CCD. In our example, this heuristic finds three global variables referenced in different functions: `MAX` is read in `full(stack *p)`, `TAM2` is read in `dequeue(queue *c, int x)` and `MAX2` is read in `isFull(queue *c)`. Then, the CCD *stack* adds *MAX*, and the CCD *queue* adds *TAM*2 and *MAX*2 in their respective list of attributes. Thus, the result of the modified CCDs are: $stack = (\{top, item[TAM], MAX\}, \{full, empty, push, pop\})$, and $queue = (\{front, rear, length, item2[TAM2], cant, TAM2, MAX2\}, \{enqueue, dequeue, isFull, isEmpty\})$

In the case of module `list.h`, this heuristic finds that `MAX3` is read in `listfull()`, and `TAM3` is read in `insertafter()` and `insertbefore()`. Then, these attributes are added to the CCD *list*. The result of this CCD is: $list = (\{itemlist[], actual, TAM3, MAX3\}, \{initlist, insertafter, insertbefore, length, destroy, getitem, listfull, listempty, moveactual\})$

3.2 Expert Mapping: Refactoring

As we stated previously, the mapping from source code to class definitions is an automatic process, but the further analysis to decide the final set of classes is based on expert knowledge and a set of refactorings done by experts manually.

As we are obtaining an object-oriented model with associations built only between variables and the CCD that *type* them, and we want to take advantages of good design principles, we could propose improvements in the obtained design. Thus, we describe different possible refactorings that can be applied. Some of them are based on the Fowler refactorings (Fowler et al., 1999) and other ones are built based on specific problems when analyzing the resulting model. Due to space limitations, we will just mention some of them, and the reader can see their application in the case study explained furtherly in this paper: *(1) Extracting common arguments from methods, (2) Extracting algorithms, (3) Abstracting an algorithm: a variant of Extracting Algorithms, (4) Relocating attributes or methods, (5) Renaming CCDs, (6) Removing CCDs*. This list is not exhaustive, and just mention the main ones applied in the case study furtherly.

4 VALIDATION

The source code of the thirteen case studies that we use to validate comes from two different sources:

- Ten case studies (*University, AssemblyLine, BallotBoxes, AirportABC, LightBulbs, Elevators, MixAdt1, MixAdt2, MovieClub, Library*) were designed with UML class diagrams and implemented in C by a group of advanced computer science students.
- Three case studies (*Calculator, Bank, Airport*) were downloaded from different Internet websites. As we did not have UML class diagram (as previous cases), they were also built by the same group of advanced university students that worked in the first set of case studies.

The classes of the UML diagram are used to perform the comparison of the original design model with the one we obtain with our approach. We have chosen the case studies from different sources, because in the case of applications implemented by university students we consider them as the experts that know the complete model and can validate our results. Then in the case of applications downloaded from Internet, we use them to show the applicability in any case study (without needing the experts to validate our results).

We have applied the M2K approach on each case study, without considering the basic modules, such as `stdio.h` or `conio.h` in our analysis. Table 1 shows an overview of the size of the case studies and the size of the class-based models before and after applying the M2K approach. We can observe the numbers of classes in the original model are less or equal than in the refactored model. Based on our detailed analysis, we observe that the original classes are kept in the refactored model.

Table 1: Number of classes in different analysis phases of each case study.

Case Study	LOC	UML classes	Initial CCDs	Refactored CCDs
University	162	1	3	7
AssemblyLine	168	3	4	3
Calculator	200	1	1	10
Bank	216	2	2	2
BallotBoxes	228	7	6	9
AirportABC	282	5	7	9
LightBulbs	285	6	4	8
Elevators	295	8	6	10
Airport	341	3	4	4
MovieClub	116	4	4	3
Library	341	6	6	6
MixAdt1	281	2	2	2
MixAdt2	273	2	1	1

Case Study: University. Due to space limitations, we show only a case study in detail. *University* is an application that takes information fron two arrays in order to obtain a summary of the information of the students. Figure 1 shows the resulting CCD-based model in this case. Now, when we analyzed this generated model, we discovered seven possible refactorings based on the obtained design:

1. **Abstracting an Algorithm.** The method `BubbleSort(STUDENT_FINAL average[], int array_size)` (mapped from `Bubblesort` function) is implementing the bubblesort algorithm to sort the students using their grades. The refactoring is to implement a more abstract sorting algorithm that could be used by any data, and then use that implementation with the student data that is used in this application.
2. **Extracting Algorithms.** There are three methods mapped from three functions that are performing different printing actions `print_best_10_averages(..)`, `print_average_per_student(..)`, `print_exams_per_student(..)`. We could extract the common behavior and refactor the different ones in several new CCDs, as in *Strategy Design Pattern*.
3. **Extracting Common Arguments from Methods.** According to ModelMapper, we have built a CCD `Student_Final` with the structure described previously, and that CCD allows us to know that the arguments of the methods of the CCD `Student.h` have that type. However, when we have a look at the methods in this last CCD, we discovered that those arguments are arrays (`STUDENT_FINAL finals[]` and `STUDENT_FINAL averages[]`). One example method is `search_exams_per_student(int code, STUDENT_FINAL finals [])`. This means that in fact these arguments are common to most functions, and they could be refactored and transformed into attributes `finals` and `averages` in the CDD abstracting the set of students final exams and averages.
4. **Relocating Attributes.** In the CCD `University.c` (mapped from the module that contains the function `main()`), we have two constants `ELEMENTS` and `STUDENTS`. As they are used in the CCD `Student.h`, we relocate them in this last one, and we map them as class variables.
5. **Renaming CCDs.** Once we have applied all previous refactorings, we can rename the CCDs according to the concept domain they represent. Thus, the CCD `Student_Final` is named as

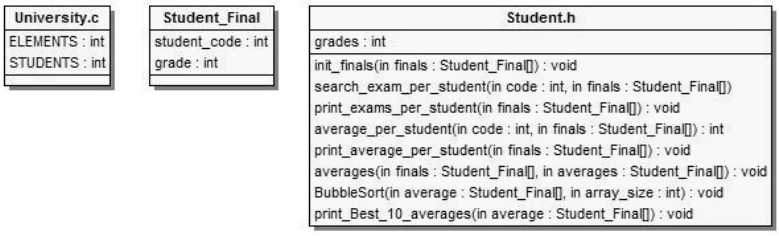

Figure 1: Case Study University. Resulting CCD-based model.

`Student` and the CCD `Student.h` is named as `Student_Set`

6. **Removing a CCD.** Given that when we have relocated the attributes of the CCD `University.c`, now this CCD is empty and can be removed from the refactored model.

With these refactorings, we obtain an improved CCD-based model (shown in Figure 2) of our application.

5 ANALYSIS: APPLYING REFACTORINGS

Once we have built the CCD-based model in the *Mapping* phase, we refactored it to improve its object-oriented design. In Table 2, the columns *Refactorings* shows the number of case studies in which the refactoring was applied, and *Percentage* shows the same information but in terms of the thirteen case studies, for example the refactoring named as *Extracting algorithms* is applied in 6 case studies, representing the 46% of them. When analyzing the frequency of each refactoring, the most applied one is *Relocating attributes or methods*. This happens because when building applications with structured paradigm, the variables are not associated necessarily to the functions use them. When M2K approach maps this structure in a object-oriented model, as a consequence the attributes and the behavior of the built CCDs are not associated correctly, and then the expert needs to relocate the model entities to the right CCD. Another most used refactoring is *Renaming CCDs*, and this happens because the names close to the domain concept can not be inferred from the code, because they are related to the domain they are representing in both the legacy code and the high-level model. Both refactorings (*Relocating attributes or methods and Renaming CCDs*) highlight an important goal of the approach: the experts do not add classes that were not mapped previously because they are identified in the automatic process of the approach. The added classes made in the refactorings *Extracting algorithms* and

Table 2: Applied Refactoring applied to case studies. #R is the number of refactorings and % is the percentage in terms of all case studies.

	#R	%
Relocating attributes or methods	8	61
Renaming CCDs	7	54
Extracting algorithms	6	46
Extracting common arguments	5	38
Removing CCDs	5	38
Abstracting an algorithm	1	8

Abstracting an algorithm focus on design classes and are not related to domain concepts of the applications.

The refactoring *Extracting algorithms*, as the previous refactoring, depends on the implementation of the C application. We applied the Strategy pattern to extract the common behavior in the CDD delegating behaviour to new classes.

The refactoring *Removing CCDs* is used to remove unneeded classes. After the automatic part of the approach, the expert receives a set of candidate classes where some correspond to the real Classes of the UML and some are discarded. The ones that are removed either correspond to empty classes or do not represent domain concepts.

Finally, the refactoring *Extracting common arguments* is a consecuence of migrating from procedural to object-oriented paradigm. In this last one, it is not needed that a method makes an explicit reference to the attributes that use in its parameters, but in C Code does it.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we propose the M2K methodology that generates class definitions from source code in C. The mapping of the legacy code is an automatic process and the further analysis to apply refactorings is done manually by experts. It differs from other existing works that it uses heuristics to look for cohesive entities and generate a set of class candidates. By us-

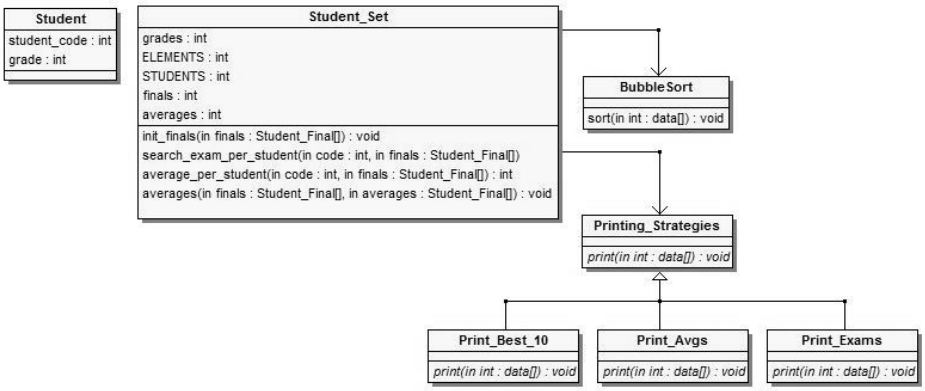

Figure 2: Case Study 2: University. Modified CCD-based model.

ing this methodology we can improve the understanding of a legacy code, propose some refactorings and generate a design document. Our future work includes to automate some refactoring activities, and to apply a ranking to improve the filtering of the generated CCDs. We will work on a new heuristic to find associations between CCD. In the case of identifying inheritance, we consider that it would be possible to propose a improvement in the *Expert mapping* phase. C language does not have statements that could be mapped as this design principle. A more complete validation with a legacy code with 20KLOC will measure the complexity and effort of refactoring the reconstructed model, and the scalability of the approach.

REFERENCES

Astels, D. (2002). Refactoring with UML. In *Proc. Int'l Conf. XP and Flexible Processes in Software Engineering*, pages 67–70. Alghero, Sardinia, Italy.

Boger, M., Sturm, T., and Fragemann, P. (2003). Refactoring browser for uml. pages 366–377.

Fanta, R. and Rajlich, V. (1999). Restructuring legacy C code into C++. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 77–85. IEEE Computer Society Press.

Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison Wesley.

G. Canfora, A. and M.Munro (1996). An improved algorithm for identifying objects in code. *Software Practice and Experience, 26(1):25-48*.

G. Sunyé, Pollet, D., LeTraon, Y., and J.-M. Jézéquel (2001). Refactoring UML models. In *Proc. UML*, volume 2185 of *LNCS*, pages 134–138. Springer-Verlag.

Garrido, A. and Johnson, R. E. (2003). Refactoring c with conditional compilation. In *ASE*, pages 323–326. IEEE Computer Society.

Nierstrasz, O., Ducasse, S., and Gĭrba, T. (2005). The story of moose: An agile reengineering environment. *SIGSOFT Softw. Eng. Notes*, 30(5):1–10.

Sahraoui, H. A., Melo, W. L., Lounis, H., and Dumont, F. (1997). Applying concept formation methods to object identification in procedural code. In *ASE*, pages 210–218.

Siff, M. and Reps, T. (1999). Identifying modules via concept analysis. *IEEE TSE*, 25(6):749–768.

van Gurp, J. and Bosch, J. (2002). Design erosion: Problems and causes. *J. Syst. Softw.*, 61(2):105–119.

Yadav, R., Patel, R. P., and Kothari, A. (2014). Reverse engineering tool based on unified mapping method (retum): Class diagram visualizations. *Journal of Computer and Communications*, (12):39–49.

Yeh, A. S., Harris, D. R., and Reubenstein, H. B. (1995). Recovering abstract data types and object instances from a conventional procedural language. In *Proc. of WCRE)*, page 227. IEEE Computer Society.

Yourdon, E. and Constantine, L. L. (1979). *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc.

Zou, Y. and Kontogiannis, K. (2001). A framework for migrating procedural code to object-oriented platforms. In *Proc. of IEEE APSEC*, pages 408–418, Los Alamitos, CA, USA. IEEE Press.