Maintenance of Object Oriented Systems through Re-engineering: A Case Study

Manoranjan Satpathy[1], Nils T Siebel[2] and Daniel Rodríguez[1]

[1]Applied Software Engineering Group
[2]Computational Vision Group
Department of Computer Science
The University of Reading
Reading RG6 6AY, England

m.satpathy@reading.ac.uk
{nts,drg}@ieee.org

Abstract

Unregulated evolution of software often leads to software ageing which not only makes the product difficult to maintain but also breaks the consistency between design and implementation. In such a case, it may become necessary to re-engineer the software so that it becomes maintainable again. In this paper, we present the case study of the re-engineering of the People Tracking subsystem *of a surveillance system written in* C++*. We discuss the problems, the challenges and the approaches taken, and we show how the re-engineered product is now better maintainable. We also discuss the generation of the relevant artefacts — from requirement document through to design document.*

Keywords. *Software Maintenance, Reverse Engineering, Re-engineering, Case Study, OO Systems.*

1. Introduction

Software maintenance is the modification of a software product after delivery. It is classified into four categories [5, p. 354–355]:

1. *Corrective maintenance* refers to modifications for correcting problems in an implementation.

2. *Adaptive maintenance* refers to modifications for adapting a product to changed environments, both software and hardware.

3. *Perfective maintenance* refers to enhancements such as making the product faster, smaller, better documented, creating a cleaner structure, and adding new functionalities because of new user requirements.

4. *Preventive maintenance* involves changes which are done aiming at preventing malfunctions and improving maintainability of the software.

Initial design of a software aims at developing a product so that modifications of the above kinds become easy to perform in terms of time and effort. Furthermore, it is expected that as a consequence of these changes, the software should preserve the original aim of the initial design. However, it often happens that modifications are done without proper software engineering principles, and therefore the initial design soon becomes unclean in the sense that its maintainability becomes costly in terms of effort and time.

Consider the example of cohesion and coupling [5, p. 309–317]. The interaction between various elements within a class should be maximised and the interaction across classes should be minimised. However, if too much of "patch work" was done in a "half-hazard" manner, then the implementation would no longer exhibit the desired levels of cohesion and coupling. Furthermore, for better maintainability, it is expected that the various artefacts of a software product — from requirement document through to design — remain consistent with each other and the implementation. However, because of bad maintenance over a long period of time, in many cases this consistency is lost. As a result the software loses traceability.

In such a case, it may become necessary to re-organise the software with the following aims:

- to recover the system artefacts from the implementation and the existing documents,

- to bring the artefacts to a consistent state, possibly through re-design or re-specification. The re-design should aim at restoring the quality characteristics of the software [9],

Proceedings of the International Conference on Software Maintenance (ICSM'02)
0-7695-1819-2/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

- and to re-structure the implementation in order to reflect the new design.

Consequently, the system is made maintainable again. This approach is often called *re-engineering*. Re-engineering is the approach of understanding the old code to keep much of it and to modify it to meet new needs [20]. This approach involves reverse engineering. Reverse engineering is the process of understanding and modifying software systems. It involves identification of the components of an existing system and the relationship among them. Furthermore, it also aims at creating high level descriptions of various aspects of the existing system [19].

The primary challenge with regards to re-engineering is to understand the existing software along with its associated artefacts. In order to accomplish this, we need two kinds of information: *static information* and *dynamic information*. Static information describes the structure of the software in relation to the source code, while dynamic information describes its run-time behaviour [16].

In this paper, we discuss the case of the People Tracking subsystem of an integrated surveillance system for underground stations. The details of this system will be presented in Section 3. We briefly discuss the evolution of the system and how it became unmaintainable over time and why it became necessary to re-design it. We describe how a re-engineering approach was carried out and what benefits were obtained from it.

The organisation of the paper is as follows. Section 2 discusses the related work. Section 3 presents the object of our case study and the reasons behind its re-design. Section 4 discusses the approach taken for the re-design. Section 5 discusses the system after the re-design, the benefits achieved and the lessons learnt from this case study. Section 6 concludes the paper.

2. Related Work

A lot of research has been carried out and presented on software maintenance and reverse engineering. We cite here those which we consider relevant to our case study.

Lientz et al [11] have surveyed 120 organisations and analysed their maintenance efforts in terms of their categories. Their observation: on average, 17.4% of the maintenance effort is corrective, 18.2% is adaptive, 60.3% is perfective and 4.1% is preventive.

Domsch et al [4] have presented a case study in object-oriented maintenance in which the text based user interface (UI) of a product, which determined the number of power supplies required for a system configuration, was replaced by a graphical UI (GUI). The additional requirements were: (i) the new software must print relevant graphical outputs to a printer and (ii) the new product must run on various 32-bit Microsoft Windows platforms. The requirements constrained the maintainer to use Microsoft specific APIs. The software engineer who developed the product was also the maintainer. The total maintenance effort was 116 man hours; 95% of the maintenance effort was perfective (development of GUI), 3.2% adaptive and 2% corrective.

One of the important issues in re-engineering is the detection and location of design flaws which prevent an efficient maintenance and further development of the system. Marinescu [13] has discussed a metric-based approach to detect such design flaws. The two most well-known design flaws are *god classes* and *data classes*. God classes are those which tend to centralise the intelligence of the system, while data classes are those which define data fields and almost no methods except some accessor methods. The author uses a metric-based approach for detecting god classes and data classes. A case study was done on an industrial project of 50,000 lines of `C++` code. The author points out that the approach was highly effective, though there are design flaws like duplicated code and the number of detected bugs in a class which would not be addressed by his approach because they rely on metrics other than the ones which can be obtained from the source code.

Refactoring [6] is a technique to correct design flaws in object oriented systems. Refactoring operations reorganise a class hierarchy by shifting responsibility between classes and redistributing instance variables and methods. Therefore it would be better if reverse engineering approaches found out which refactoring approaches have been applied, for understanding system evolution. Demeyer et al [3] discuss four heuristics for identifying various types of refactoring operations which were applied to the past versions. The authors deal with the following three types of refactoring: (i) splitting methods into smaller chunks to separate common behaviour from the specialised parts so that subclasses can override, (ii) moving functionality to a newly created sibling class and (iii) insertion/removal of classes from a class hierarchy and redistribution of their functionality.

3. The Object of our Case Study

The system studied for this article is the People Tracking subsystem of `ADVISOR`[3], an integrated system for automated surveillance of people in underground stations (see Figure 1). `ADVISOR` is being built as part of a European research project involving 3 academic and 3 industrial partners. The task of the People Tracker is to automatically analyse images from one or many camera inputs. The system has to detect all the people in the image and track them in realtime as a stream of video images is continuously fed

[3]Annotated Digital Video for Intelligent Surveillance and Optimised Retrieval

IEEE COMPUTER SOCIETY

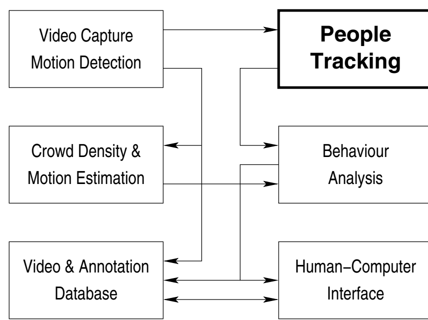

Figure 1. ADVISOR System Overview

to the system. The image in Figure 2 shows an example of the visualised output from the People Tracker.

3.1. Brief History

The original People Tracker was written at the University of Leeds in 1993–1995 using C++ running under IRIX on an sgi. It was a research and development system and a proof of concept for a PhD thesis [2]. The main focus during development was on functionality and experimental features which represented the state-of-the-art in people tracking at that time. No software design technique was employed during the code development. The only documentation generated was a short manual on how to write a program using the People Tracking module.

In 1995–1998 the code was used in a collaboration between the Universities of Leeds and Reading. The software was adapted at the University of Reading to inter-operate with a vehicle tracker which ran on a Sun/Solaris platform [15]. Only little functionality was changed and added during this time and no new documentation was created.

Starting in 2000, the People Tracker has been changed for its use within the ADVISOR system shown in Figure 1. This new application required a number of major changes at different levels. This article focuses on the changes carried out so far within ADVISOR.

Figure 2. People Tracking Results

Table 1 shows a brief summary of the characteristics of the People Tracker at different stages of the project, starting from January 2000. These metrics were obtained using the CCCC (C and C++ Code Counter) tool by Littlefair [12]. The size is given in Lines of Code (LOC) including comment lines. The original version (as of January 2000) is referred to as "PT_0".

Version	LOC	Classes	Methods	Global Func.
PT_0	38,520	154	1,177	271
PT_1	45,762	191	1,363	286
PT_2	51,435	183	1,431	25
PT_3	40,902	178	1,437	24

Table 1. Versions of the People Tracker

3.2. Motivation for Re-design

The planned use of the People Tracker within the ADVISOR System carried a lot of requirements that could not be met by the original implementation. Most of the new requirements arose from the fact that the People Tracker would now have to be part of an integrated system, while earlier it was standalone. The use of the People Tracker within the ADVISOR system also meant moving it "from the lab to the real world" which necessitated many changes. Figure 1 shows how the People Tracking subsystem is connected to the other components of the ADVISOR system.

The **new requirements** for the People Tracker were:

- The People Tracker has to be fully integrated within the ADVISOR system.
- It has to run multiple trackers for video input from multiple cameras (original software: one tracker, one camera input).
- The ADVISOR system requires the People Tracker to operate in realtime. Previously, the People Tracker used to read video images from hard disk which meant there were no realtime requirements.
- Within ADVISOR, the People Tracker has to run autonomously without requiring input from an operator.
- The software has to be ported from sgi to a standard PC running GNU/Linux to make economical system integration feasible.

The status of the existing People Tracker was evaluated in relation to the new requirements. It was observed that the system had significant deficiencies which hindered the implementation of the required new functionality:

IEEE COMPUTER SOCIETY

- Heavy use of global variables and functions meant that multiple cameras could not be used.
- Excepting a few pages of operating manual, no other documentation was available.
- Though the code was written in C++, it made a limited use of object oriented features.
- There were very little comments in the source code which made it difficult to read.
- The names of some of the classes and methods were misleading.

4. Approaches Taken for the Re-design

4.1. Stages of Work

Figure 3 shows the sequence of steps which were followed to re-engineer the People Tracker. The original People Tracker is marked as "PT_0", which was running on an sgi platform. In the first phase of the work the software was ported to a PC running GNU/Linux. The approach taken was "take code and compile", replacing the non-existent functions in the process. PT_0 contained calls to sgi video hardware and to a mathematical library which did not exist on the PC.

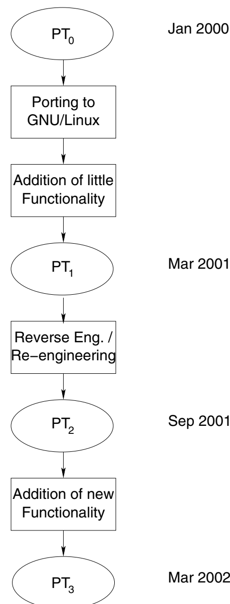

Figure 3. Sequence of Work

Once the porting was complete, attempts were made to incorporate new functionality to the code. These were input/output data format changes. Added functionality included the capability to read XML files which adhered to a given XML Schema [21] and to decode JPEG images. We refer to the product at this stage as "PT_1". The characteristics of PT_1 have again been shown in Table 1.

While incorporating the new functionality the deficiency of the software became evident. These deficiencies were discussed in section 3.2; some important influences of these were:

- As the code was badly documented it was difficult to understand.
- The lack of structure in the code (many global variables etc) meant that the collateral effects of changes were not localised.

At this stage two options were considered: (i) to re-engineer the system (ii) to develop a new system from scratch. For the following reasons it was decided to follow the re-engineering approach:

- The new developers did not have sufficient domain knowledge to develop a new system at that time, as they were learning the system in an incremental manner. The original developers were not available, and no useful documentation existed.
- It was considered necessary to have a prototype ready as soon as possible.

The aim of the reverse engineering/re-engineering step was (i) to understand the program and the design, (ii) to find and correct design flaws and (iii) to recover all the software engineering artefacts like the requirement and the design documents. From the source code, the class diagram was obtained by using the tool Rational Rose 2000e [14]. The analysis of the class diagram revealed that many class names did not reflect the inheritance hierarchy, a number of classes were found to be redundant, and many classes did have duplicated functionality. The following correctional steps were performed next:

- Redundant classes and some "dead code" [1] were removed.
- Global variables and functions were eliminated and their functionality distributed into both existing and new classes. Exceptions were global helper functions like min(), max() etc which were extracted and moved into one C++ module.
- Many refactoring techniques [3] were applied, such as:
  - Filtering out functionality duplicated in similar classes and moving it into newly created base classes.
  - Re-distribution of functionality between classes and logical modules.

IEEE COMPUTER SOCIETY

  - Re-distribution of functionality between methods.
- Consistent renaming of file names to reflect classes defined in them.
- Meaningful names were given to classes and methods.
- PT_1 contained many class implementations in the header files; they were moved to the implementation (`.cc`) files.
- Assertions [7, chap. 6] were introduced at strategic points in the existing code and in all of the new code.
- From both *static analysis* and *dynamic analysis* [16], a requirement document and the UML artefacts like the Use Case, component, and package level sequence diagrams [17] were obtained. The UML diagrams have been shown in Figures 6 through 8 respectively.

During this time, about 1 man month of testing was done to show that the modifications and additions were functioning correctly. The product after the reverse/re-engineering step is referred to as "PT_2". The characteristics of PT_2 can be seen in Table 1.

In the final step, the remaining part of the required new functionality was incorporated into the re-engineered product. This includes the addition of separate processing threads for each video input, addressing the synchronisation and timing requirements etc. A newly created master scheduler manages all processing in order to guarantee realtime performance with multiple video inputs. We will refer to this version of the People Tracker as "PT_3". Table 1 shows the characteristics of PT_3. This version of the People Tracker incorporates most of the functionality needed for its use within ADVISOR and improvements to the people tracking algorithms which make it appropriate for the application [18]. The module has been validated against test data. Currently (March 2002), the final stage of system integration is being undertaken.

4.2. Maintenance Effort

The maintenance effort (in man months) in relation to the various stages in Figure 1 can be found in Table 2. Considering the size of the project, a total of 26 man months of maintenance effort might seem a lot. The reason is that the developers were doing porting and re-engineering for the first time. Hence they took some time to understand the underlying concepts. Table 3 shows the percentage of maintenance effort by various maintenance categories. For comparison purposes, we also give the average effort observed by Lientz, Swanson and Tompkins (LST) [11]. We can infer the following:

Stage	Porting	Little Func.	Re-Eng.	New Func.
Effort	8 MM	4 MM	8 MM	6 MM

Table 2. Distribution of Maintenance Effort

- Little corrective effort was necessary as the software was running without showing any significant errors at the start of the project.
- As the software needed to be ported to a different platform, the adaptive effort is higher than average.
- While the sum of perfective and preventive maintenance effort were found to be similar to the LST average, there is a clear shift towards preventive maintenance. The reason is that a lot of effort was put into improving the maintainability of the software, like incorporating assertions into the code.

Category	Effort	Average (LST)
Corrective	8 %	17.4 %
Adaptive	31 %	18.2 %
Perfective	38 %	60.3 %
Preventive	23 %	4.1 %

Table 3. Maintenance Effort by Category

4.3. Code Size

Figure 4 shows the measures of code size as the software was passing through the various stages shown in Figure 3. Up to PT_1, functionality was simply added without removing much unused functionality. During the re-engineering phase, the code size increased. This is because the code was made more modular and scalable through the introduction of new middle layers, and documentation was added to the code. Also, unused functionality was identified and marked as such. However, almost up to version PT_3, while new functionality for ADVISOR was added, most of the unused legacy functionality was kept in case it would be needed again. When it was finally removed in the last version examined here, one can see the sharp decrease in code size.

5. Further Analysis and Discussion

5.1. Generated Artefacts of the People Tracker

As a part of the re-engineering process, various artefacts were recovered from the code, and they were upgraded to reflect the code of the latest version of the People Tracker. In addition to the requirement document, the other remaining artefacts were mostly UML diagrams. The initial class

IEEE COMPUTER SOCIETY

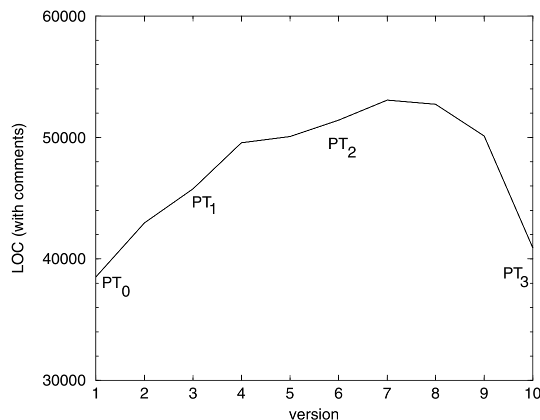

Figure 4. Lines of Code over Time

diagram which was obtained using the Rational Rose was the primary source of all activities. Both static and dynamic analysis were employed to obtain the remaining UML diagrams. We now present a brief description of these:

- Use Case diagram(s): We obtained two Use Case diagrams: one when the People Tracker works standalone, and the other when it is seen as a subsystem of the `ADVISOR` system. They have been shown in Figures 5 and 6 respectively. The former corresponds to the Use Cases as seen by the developer, whereas the latter corresponds to the Use Cases after the subsystem is integrated with rest of the subsystems of the `ADVISOR`.
- Class Diagram: The initial class diagram was upgraded to incorporate the new class structure after refactoring and other transformations. We have not shown this diagram here because of its size. Figure 7 shows the package diagram.
- Sequence Diagram: Figure 8 shows the sequence diagram of the People Tracker at a higher level.

5.2. Experience with the Re-engineered software

The re-engineering of the People Tracker (refer to Figure 3) made addition of new functionality easier. We illustrate this by a few examples:

- All classes handling the data and tracking functionality for one camera (ie. one video input) were encapsulated within one `Camera` class. As a result, the handling of multiple cameras simply amounted to creating multiple instances of the `Camera` classes. The original version (PT_0) of the software handled functionality like the video input, tracking etc. directly in the `main()` program, and only one video input was implemented.
- On a larger level, all `Camera` classes within one area of the underground station are contained within one `PeopleTracker` class. Hence the system design allows for scaling of the system simply by instantiating more than one `PeopleTracker` class. This was done keeping the scalability aspect of the design in mind, although this functionality is not required at present.
- In the new design, all tracking results are contained within one `Results` class. As a consequence, the new requirement to write out the results in XML format was a very localised operation. Additionally, changing the design to reflect the new functionality was straightforward.

5.3. Improved Maintainability

The re-engineering of the software has improved its maintainability. We show this by considering each of the maintainability categories.

Corrective Maintenance The software now uses assertions, hence it is easier to find bugs at an early stage. Also, the new design means that malfunctions of particular functionalities can now be easily localised and corrected.

Adaptive Maintenance The new software is not dependent on any particular hardware any more. The code now strictly follows the ANSI/ISO `C/C++` standard, compliant with ISO/IEC 9899:1999(E) [10], as well as IEEE POSIX 1003.1c-1995 extensions [8] for all multi-threading functionality. As a result, future porting of the code should be easy.

Perfective Maintenance The software is now better documented and it has a clean structure. Furthermore, all the artefacts of the system are in a consistent state. Hence, adding new functionality is easier, as already experienced, and discussed above.

Preventive Maintenance Assertions, which are now used, help preventing malfunctions. Additionally, the re-engineering and the new design have improved the maintainability of the software.

5.4. Personnel Factors

The skills of the personnel involved in the work described in this article are as follows:

1. A Ph.D. student having academic knowledge of software engineering and OOP but little experience in software maintenance. He was the team leader and he himself did 60% of the whole task.

IEEE COMPUTER SOCIETY

2. A Ph.D. student having academic knowledge of software engineering, moderate level of programming experience and no experience in maintenance. He did 15% of the task.

3. Two undergraduate students having no knowledge of software engineering but good programming experience. They did 20% of the task.

4. A senior researcher having no knowledge of software engineering and a moderate level of programming experience. He did 5% of the work described here.

The lack of maintenance experience of all the people involved was a prominent factor in making the whole process inefficient. Instead of making an initial work schedule, they started to implement new functionality at too early a stage. Only after implementing a little functionality, they inferred that re-engineering of the software was necessary. As a result, precious time was lost.

5.5. Lessons Learned

1. Any kind of maintenance activity, especially re-engineering, must be preceded by adequate planning. In the present case, initially attempts were made to incorporate the new functionalities and only when the attempts failed the maintainers decided to re-engineer the software.

2. Some training should be given to maintainers, especially when they are not experienced in doing maintenance, for recognising that certain code is not maintainable. In the present case, the maintainers took time to learn about the extent to which the code was not maintainable. At this stage, some tool support could aid in determining this aspect of the code. The tool could use indicators like the number of global variables and global functions, lack of documentation etc.

3. Initially, the maintainers felt a strong resistance against re-engineering the software. Even when they found out that adding new functionality would be difficult with the current state of the code, they hesitated to take the step to re-engineer the software because this was not a part of their given task. This seems to be a common problem within software projects: a tradeoff between short-term goals like adding functionality to the code on one side, and long-term effects like having a product which is better maintainable. While developers are more likely to determine the necessity to re-engineer a product, managers tend to resist it and focus on short-term goals. Re-engineering takes up a lot of time and resources in the short term, although in the long term it might save time as maintainability increases.

4. The knowledge of a precise model of re-engineering process should be in place before initiating the re-engineering approach. In the present case, the trial-and-error approach consumed a lot of precious time. A correct design should be a pre-requisite to implementing new functionality. Design documents and other artefacts should always remain consistent with the source code.

5. In the present case, refactoring transformations were applied by locating problem spots manually. Tool support could have been used to recognise bad classes (for instance, *god classes* and *data classes* [13]).

6. Developers should stick to standard languages without using language extensions. Similarly, the product should not depend on specific hardware to run. If the use of software- or hardware-specific functions are required, eg. for optimisation purposes, they should be isolated and adequately documented so as to ease future porting operations.

7. It was observed that domain knowledge was not a pre-requisite for performing the porting task. So far as re-engineering was concerned, very little domain knowledge was required for performing the static analysis, whereas dynamic analysis required significant amount of domain knowledge. Furthermore, a moderate level of domain knowledge was necessary while adding new domain specific functionalities.

8. The maintainers of the People Tracker had occasional short discussions (over lunch table or during coffee hours) with experienced software engineers in the university. They are of the opinion that this guidance increased their awareness of technical aspects of software maintenance. We therefore recommend the consultation of an experienced software engineer before and during the maintenance task.

6. Conclusion

In this paper, we have presented the case study of the People Tracking subsystem of the integrated system `ADVISOR` for the surveillance of people in an underground station. The medium sized People Tracker was originally developed in a University environment. We have described how the module was re-engineered so that it could be used in the "real world" as a subsystem of `ADVISOR`. The team leader of the maintenance operation is a co-author of this article and he has maintained the log of all the maintenance activities, from beginning to end. None of the people involved in the re-engineering task had any maintenance experience and therefore the whole process was carried out in

IEEE COMPUTER SOCIETY

a suboptimal manner. The people involved took occasional help from experienced software engineers and these, they believe, provided important guidance when it came to software engineering aspects of the work,

The maintainers successfully re-engineered the product and upgraded it with new functionality. The product has been found to be satisfactory by the project partners and it is now in the final stage of its integration with the other subsystems of ADVISOR. Furthermore, the maintainers are satisfied with the work they have done. Initially there was no documentation available with the software, but all UML artefacts were generated from the source code. Our analysis shows that the re-engineering process has been effective in achieving a high level of maintainability.

The main observations of this case study were:

- Tool support or expert advice could aid in identifying that a piece of software is not maintainable, to help in the decision whether a re-engineering process needs to be carried out.

- Increasing the maintainability of software by re-engineering techniques has made the addition of new functionality more efficient.

- Surprisingly little domain knowledge was necessary for porting or re-engineering the software. Domain knowledge was mainly needed to extract dynamic information from the code, and partly during the addition of new functionality. At this stage, the availability of design documents can to a certain extent make up for missing domain knowledge.

Acknowledgements

The authors wish to thank Steve Maybank and Rachel Harrison for making this research possible. The work was supported by the European Union (grant ADVISOR, IST-1999-11287) and the EPSRC (grant EMPAF, ER/L87347).

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] A. M. Baumberg. *Learning Deformable Models for Tracking Human Motion*. PhD thesis, School of Computer Studies, University of Leeds, October 1995.

[3] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2000)*, pages 166–177, October 2000.

[4] M. L. Domsch and S. R. Schach. A case study in object-oriented maintenance. In *Proceedings of the 1999 International Conference of Software Maintenance (ICSM '99)*, pages 346–352, August 1999.

[5] N. E. Fenton and S. L. Pfleeger. *Software Metrics*. PWS Publishing Company, 2nd edition, 1996.

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

[7] D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[8] IEEE Standards Association. *IEEE POSIX 1003.1c-1995*, 1995.

[9] International Organization for Standardization, Geneva, Switzerland. *ISO/IEC 9126*, 1991.

[10] International Organization for Standardization. *ISO: Programming languages — C. ISO/IEC 9899:1999(E)*, 1999.

[11] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software. *Communications of the ACM*, 21(6):466–471, June 1978.

[12] T. Littlefair. *An Investigation into the Use of Software Code Metrics in the Industrial Software Development Environment*. PhD thesis, Faculty of Communications, Health and Science, Edith Cowan University, 2001.

[13] R. Marinescu. Detecting design flaws via metrics in object-oriented systems. In *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Language and System (TOOLS USA 2001)*, pages 173–182, July/August 2001.

[14] Rational Software Corporation, Cupertino, CA. *Rational Rose 2000e*, 2000.

[15] P. Remagnino, A. Baumberg, T. Grove, T. Tan, D. Hogg, K. Baker, and A. Worrall. An integrated traffic and pedestrian model-based vision system. In A. Clark, editor, *Proceedings of the Eighth British Machine Vision Conference (BMVC97)*, pages 380–389. BMVA Press, 1997.

[16] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *Proceedings of the 1999 International Conference of Software Maintenance (ICSM '99)*, pages 13–22, August 1999.

[17] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[18] N. T. Siebel and S. Maybank. Fusion of multiple tracking algorithms for robust people tracking. In A. Heyden, G. Sparr, M. Nielsen, and P. Johansen, editors, *Proceedings of the 7th European Conference on Computer Vision (ECCV 2002)*, volume IV, pages 373–387, May 2002.

[19] R. C. Waters and E. Chikovsky. Reverse engineering progress along many dimensions. *Communications of the ACM*, 37(5):23–24, May 1994.

[20] B. W. Weide, W. D. Heym, and J. E. Hollingsworth. Reverse engineering of legacy code exposed. In *Proceedings of the 17th International Conference on Software Engineering (ICSE-17)*, pages 327–331, April 1995.

[21] World Wide Web Consortium (W3C). *XML Schema Part 0: Primer. W3C Recommendation*, 2001.

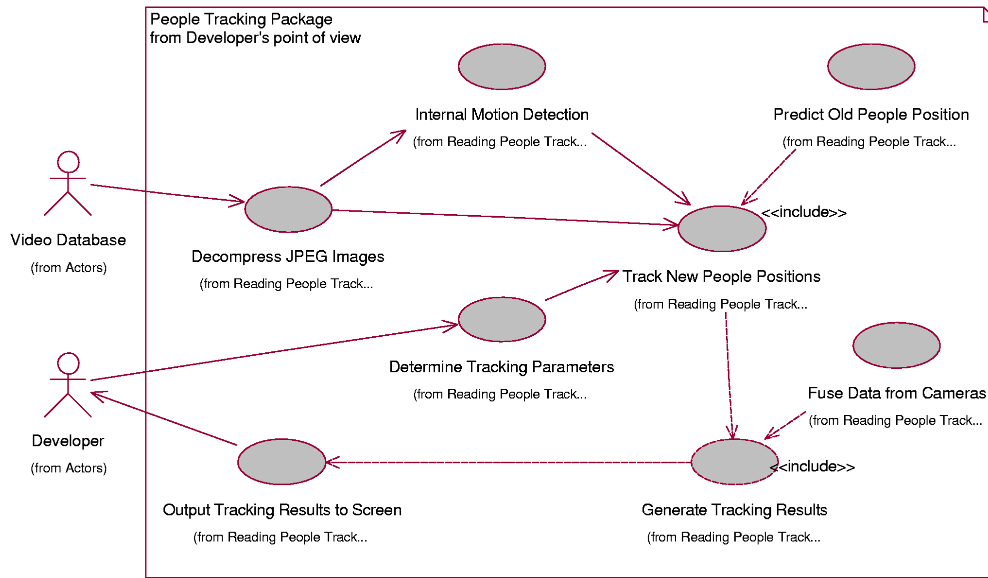

Figure 5. Use Cases for the People Tracker (in Standalone/Development Mode)

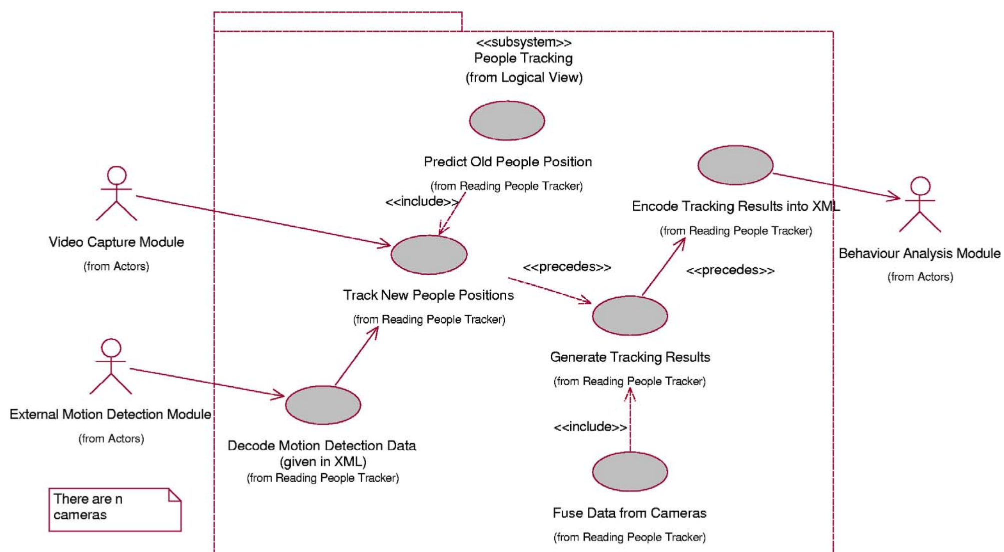

Figure 6. Use Cases for the People Tracker (as a Subsystem of `ADVISOR`**)**

IEEE COMPUTER SOCIETY

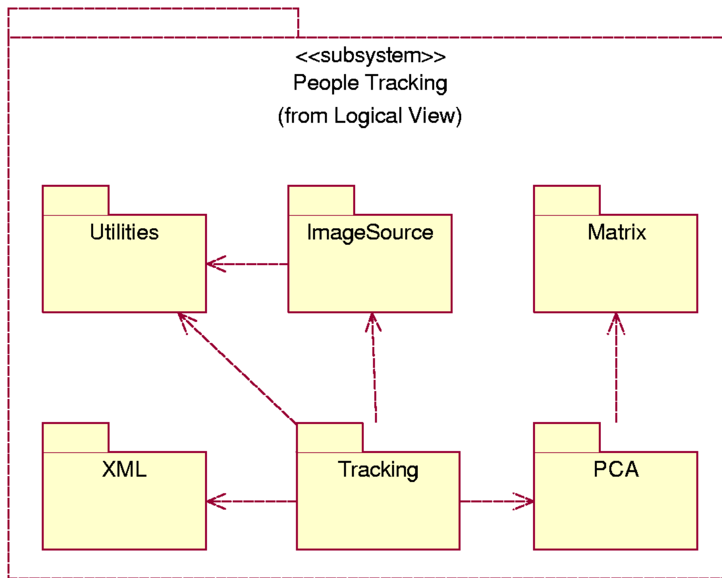

Figure 7. Software Packages of the People Tracker

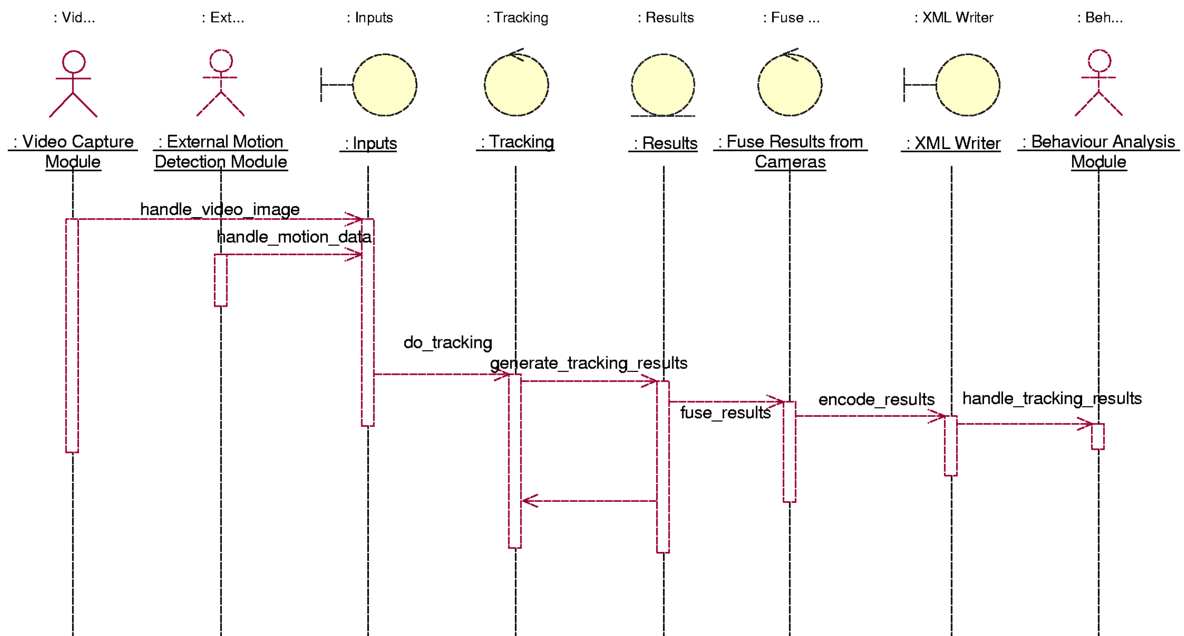

Figure 8. ADVISOR **Sequence Diagram (at a Higher Level)**

IEEE COMPUTER SOCIETY