TAMPERE UNIVERSITY OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING

SOFTWARE SYSTEMS LABORATORY

TIMO RAITALAAKSO

Dynamic Visualization of C++ Programs with UML Sequence Diagrams

Master of Science Thesis

Subject approved by departmental council on 15.3.2000

Examiner: Prof. Kai Koskimies

Foreword

I have written this thesis for Software Systems Laboratory in Tampere University of Technology. The thesis was written as a part of ATOS research project (Advanced Tools for Object-oriented Software development). ATOS project is financed by TEKES (Finnish Center for Technological Development), Nokia, Acura, Neles Automation, Plenware and Sensor SC.

Thanks to the examiner Kai Koskimies, ATOS project, and all the proof-readers for the effort put to improve this thesis, and Piia for patience.

Tampere 7.12.2000

Timo Raitalaakso

Mekaniikanpolku 8 B 19

33720 Tampere

rafu@iki.fi

Tiivistelmä

TAMPEREEN TEKNILLINEN KORKEAKOULU

Sähkötekniikan osasto

Ohjelmistotekniikka

RAITALAAKSO, TIMO:

C++ ohjelmien dynaaminen visualisointi UML sekvenssikaaviolla

Diplomityö, 54 sivua

Tarkastaja: Prof. Kai Koskimies

Joulukuu 2000

Avainsanat: dynaaminen, olio-ohjelma, takaisinmallinnus

Tässä diplomityössä annetaan yleiskuva ohjelmistojen takaisinmallinnuksesta ja erityisesti dynaamisesta takaisinmallinnuksesta. Dynaamisessa takaisinmallinnuksessa keskitytään olio-ohjelmistojen toiminnan kuvaamiseen UML:n sekvenssikaaviolla, sekä tapoihin, joilla C++ ohjelmasta saadaan ajoaikaista tietoa olioiden välisestä vuorovaikutuksesta.

Työssä esitellään Windows NT 4.0 ympäristössä toimiva DYNO järjestelmä. Sillä saadaan Microsoft Visual C++ 6.0 ympäristössä toteutetuista ohjelmista instrumentoimalla ja uudelleen kääntämällä tuotettua ajoaikaista tietoa olioiden vuorovaikutuksesta. Tämä esitetään Nokia tutkimuskeskuksessa toteutetussa TED mallinnusohjelmistossa sekvenssikaaviona. DYNOn osat on toteutettu COM komponentteina. COM komponenttien käytössä pyrkimyksenä on saada mahdollisesti uudelleen käytettäviä osia esimerkiksi vastaavan ohjelmistokokonaisuuden toteuttamiseksi Java- tai muille olio-ohjelmistoille.

Abstract

TAMPERE UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering

Software Systems Laboratory

RAITALAAKSO, TIMO:

Dynamic Visualization of C++ Programs with UML Sequence Diagrams

Master of Science Thesis, 54 pages

Examiner: Prof. Kai Koskimies

December 2000

Keywords: dynamic, object-oriented, reverse engineering

An overview to reverse engineering and especially dynamic reverse engineering of object-oriented software systems is given in this thesis. The focus in dynamic reverse engineering is in describing the behavior of object-oriented systems with UML sequence diagrams, and ways to produce runtime information about interactions between objects in a C++ program.

In this thesis a DYNO system is introduced. DYNO collects information about runtime object interactions needed to produce sequence diagrams. With DYNO it is possible to analyze programs that are generated with Microsoft Visual C++ 6.0. DYNO operates in Windows NT 4.0 environment. Produced sequences are visualized in TED software-engineering environment. DYNO is implemented with COM components. The aim in using COM components is to offer a possibility to get reusable parts to develop similar systems to other object-oriented languages like Java.

Contents

Figures

1 Introduction

Are you developing, maintaining or debugging a software system? Are you trying to reuse a framework but you don't have a clue how the framework operates? Lack of documentation might drive you to create a system of your own, similar to an existing system. These are only exaples of questions which could motivate using reverse engineering approaches and tools.

New requirements or the need to change platform of a software system will force you to redesign a program. Legacy systems are complex. Thus, it is expensive to recreate a similar system. Many problems have been solved during system evolution process of the currently running version. There is no need to create the whole system all the way from the beginning. Often the only documentation of a program is the code and those forgotten thoughts inside the head of the program creator who wrote the system many years ago. Reverse engineering tools are currently available, for example, for easing program comprehension and creating documentation.

Both static and dynamic information are useful in understanding existing software systems. Static information describes the structures of a system as they are written in the source code. Dynamic analysis is the process of analyzing the runtime behavior of software. Sometimes you need to run the code to understand it. This is in fact a necessity in object-oriented programs. Object creation, object deletion/garbage collection and dynamic binding make it very difficult and often impossible to understand the behavior by just examining the source code.

Many of the currently available reverse engineering tools are created for conventional functional-based programming languages like Cobol and C. That is because there still are legacy systems written in those languages. However, because of the popularity of object-oriented programming languages, the amount of object-oriented legacy systems has increased during the past few years. This will most probably be the trend in the future as well.

The focus of this thesis is on a small part of dynamic analysis, in creating sequence diagrams for visualizing the behavior of object-oriented programs. Sequence diagram is a

figure that visualizes the behavior of a program. Sequence diagram is one of the behavioral diagram types of Unified Modeling Language (UML)[Rat99b][RJB99]. In a sequence diagram, the trace of messages between particing objects is described.

Reverse engineering is needed mainly for large systems. There is a lot of information even in a static model of a large program. In a typical dynamic trace of a program there is many times more information. The main problem in visualizing large systems is producing readable figures that tell enough about the system in question. Understanding the dynamic information usually requires vision and understanding of the static structure.

In this thesis sequence diagrams are visualized by TED 1.0 design tool [Wik98]. TED is a graphical software-engineering environment that is based on UML model. TED is designed to work only in the Windows NT 4 environment. TED uses Object Store 5.1 SP 2 as a repository for the stored objects. TED is based on component architecture and is created using Microsoft COM components.

The DYNO system is described at the end of this thesis. DYNO produces sequences from existing programs. The source code of the object system is needed because tracking runtime events requires instrumentation of code. DYNO is created mainly using COM components with wishes of reuse of some parts of DYNO, e.g. when extracting parts of DYNO to other languages than C++.

The compiler used in DYNO is Microsoft Visual C++ 6.0 because the COM support is not available in the design environments of other manufacturers. The object system from which sequence diagrams are needed should also be manufactured with Microsoft Visual C++.

This thesis is structured as follows: in Chapter two an overview is given for reverse engineering and dynamic analysis. A view into sequence diagrams is given in Chapter three. Aspects to be considered in visualization are discussed in Chapter four. In Chapter five instrumentation as a part of dynamic reverse engineering and other methods to collect runtime information are discussed. The actual runtime sequence diagram producer, DYNO, is described in Chapter six. Conclusions and future development tasks for developing DYNO are discussed in Chapter seven.

2 Background

2.1 Reverse Engineering

The goal of reverse engineering [Mül00] is to analyze software systems so that the software is more understandable for evolution purposes. The evolution process includes maintenance, migration and reengineering. The terms reengineering or round trip engineering are used when reverse engineering is used together with conventional forward engineering. In reverse engineering the subject system in use is not changed. It is only studied and analyzed. In figure 2.1 the phases of software development process in conventional forward engineering are described. In reverse engineering the steps are taken backwards.

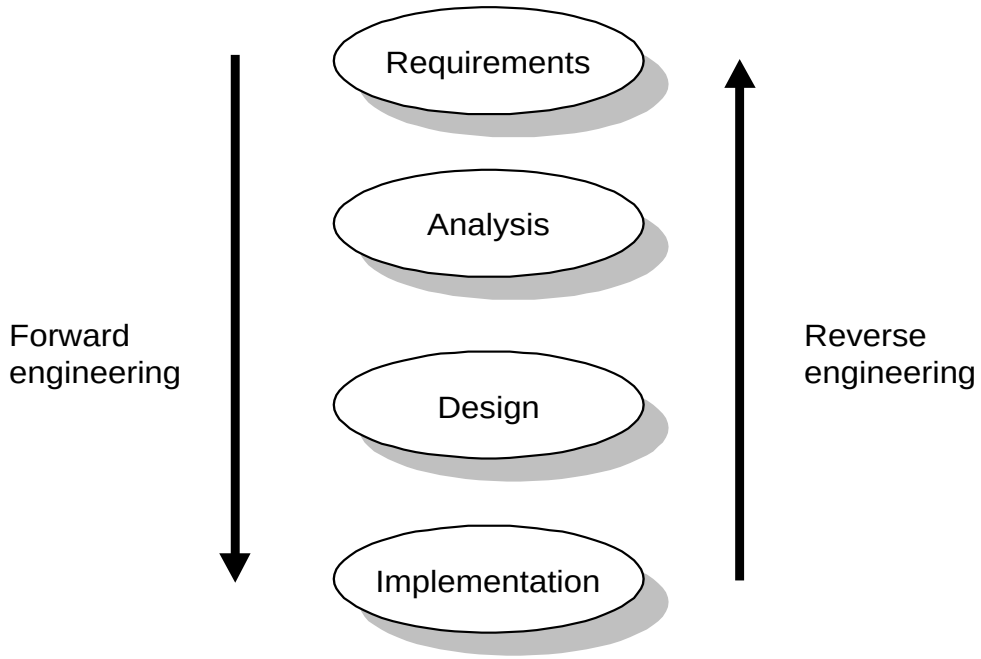

Figure 2.1: Round trip engineering

In a reverse engineering step from implementation to design parsing is used. The design of a software system is stored in a software repository. Different kinds of analysis can be made based on design using metrics and visualization. Documentation is used to define requirements of a software system. Next there are described terms that are used in reverse engineering.

Parsing

Every programming language has a grammar. Based on this it is possible to create a parser specific to a programming language. By using a parser it is possible to collect information straight from the source code. Parsers are used traditionally in compilers.

Software repository

Software repository includes metadata of a program. Metadata is information about the structure of the program. This includes, for instance, variables and dependencies. In short, metadata is data about data. Software repository might also include version handling and documentation. Repository browsing gives an opportunity to create views of a program.

Static and dynamic analysis

Static analysis is the process of analyzing software without executing it. Static analysis includes syntactic analysis, type checking and inference, control flow analysis, data flow analysis, slicing, reachability, complexity measures, static calls and structural analysis. Different program dependency graphs are used for the various static analyses like abstract syntax tree for syntactic analysis and control flow graphs for control flow.

Dynamic analysis is the process of analyzing the runtime behavior of a software system. Dynamic analysis is presented in Section 2.2 in more detail. Differencies and similarities in static and dynamic models are discussed in Section 2.3.

Pattern recognition

There are commonly used ways to implement certain basic structures of a program. These are called design patterns [GHJV95]. A programmer uses a design pattern when he is familiar with it. Some times a programmer writes code and uses structures that are similar to well-known design patterns. Identifying these patterns in an existing program is one way to help understanding a program. DP++ [Ban98] is an automated pattern identification tool.

Metrics

All kinds of things can be measured. Lines of code is the most used metrics. Also complexity of functions is measured in different ways. From dynamic point of view, memory consumption and the execution time of an algorithm are the most important things to be measured. A metric gives a numerical value describing a specific aspect of the program. Metrics gives an opportunity to find bottlenecks in a program. Which part of a program takes up unnecessarily great amount of memory, or in which part of a program time is wasted?

Software visualization

Reverse engineering gives a possibility to create different kinds of views on the structure of a program or its execution. Software visualization helps to understand a program. In forward engineering figures are used to visualize a program. In reverse engineering, it is possible to generate alternative views in addition to existing ones. By using these it is possible to correct existing documentation, and understand the program's behavior better. For a program maintainer it is possibly essential to see some vital part of a program in alternative ways than the source code.

Documentation generation

Documentation generation helps to keep the documents of a software product to be up to date. The source code documentation and the program documentation are often similar. Problems arise when the code is modified and the documentation is not updated and vice versa. The documentation can be inside the source code and an external documentation is produced from there like it is done in Javadoc [Sun00]. Javadoc is a tool that parses the declarations and documentation comments in a set of Java source files and produces a set of HTML pages describing the classes, inner classes, interfaces, constructors, methods, and fields.

Reverse engineering allows us to enhance understanding and cope with the complexity of a program. In large programs, it is necessary to produce higher-level abstractions to make the program more understandable. It is possible to generate alternative views of a program by using reverse engineering approaches and tools. This way it is possible to create documentation, which may not have been even produced in many cases. Lost information not ever documented can be recovered. And even if there is documentation, it is often not

updated. The documentation may describe earlier versions of a program or include characteristics that are not yet implemented.

With reverse engineering tools a programmer might find ways to create a better architecture. After redesigning and implementing the new architecture with parts of old code, or after adding new features into an existing system unwanted behavior might come up. An old function that is used does actually more than it was believed to do or the updated function does not manage to comprehend all of its previous tasks. After adding new code to an existing system dead code often appears. Dead code is a piece of code that is never executed. Detecting these side effects might be extremely difficult without metrics, traces, and/or debugging. The results of reverse engineering facilitate reuse and they can also be used to populate a software repository.

The use of reverse engineering methods requires that actual tools are provided. Tool integration is one way to provide reverse engineering facilities to the users. For instance, current development environments like Visual C++ or Borland C++ Builder provide reverse engineering tools as a part of the product. Information about the program structure is collected. When an existing C++ -file is opened classes are recognized and visualized. Design tool Rational Rose has tools for round trip engineering for C++ [Rat99a] and Java [Rat98]. In these, reverse engineering tools are used together with code generation.

If you do not know why you are reverse engineering or you do not know how to use the information produced, do not do it. It is easy to invest too much effort to achieve a result that is not useful. Planning is needed. If you put garbage in, garbage gets out.

2.2 Dynamic Analysis

It is often difficult to understand the behavior of a complex program by examining its static model. Usually visualization of the dynamic behavior of the program is needed to give more information of the program. Before the dynamic analysis you need to have a vision of the static structure of the program. Dynamic visualization gives also an opportunity to look for bottlenecks and false behavior of a program. Monitoring what happens during runtime is called dynamic analysis.

Debugging

Debugger allows the user to control the execution of code and examine program status, values of variables during the execution of the program. With a debugger it is possible only to examine values in break points marked into code beforehand. With some debuggers it is possible to catch exceptions to figure out the wrong behavior of a program. Most compiler products today have their own debugger solutions included.

Coverage analysis

Coverage analysis is done for studying subsets of code executed for a given input dataset. Coverage analysis is good for test coverage. If a product needs to be properly tested there should be test datasets that allow branches, paths, statements, functions, etc. to be covered (all executed at least once).

CTC++ [Test00] test coverage analyzer is useful for coverage and profiling analyses. Profiling analysis is for identifying frequently executed code. This way it is possible to find and isolate bottlenecks and performance hotspots of a program. CTC++ instruments the code. After running the instrumented program with test data it produces reports with (how many times visited):

- Function coverage (functions called)
- Decision coverage (condition expressions true and false in program branches)
- Statement coverage (statements executed; concluded from function and decision coverage)
- Condition coverage (elementary conditions true and false in condition expressions)
- Multi condition coverage (all possible ways to evaluate a condition expression executed)
- Interface coverage (CTC++'s measure related to testing of inherited C++ classes)

Memory analysis

Memory analysis is a way to study the dynamic memory usage of a program. The goal is to detect memory access or usage problems.

Memory access problems include:

- using uninitialized memory
- accessing memory not allocated to a program
- writing to freed memory
- reading/writing beyond the end of an array
- using a pointer that does not point to something valid
- dereferencing a NULL pointer
- freeing non-heap or unallocated memory
- overflowing the stack

Memory usage problems include:

- failing to reclaim or free unneeded memory
- memory leak (allocated memory but no active pointer to it)

Tutnew library [Rin00] with C++ programs is useful for avoiding memory leaks and detecting memory corruption. It is not actually a reverse engineering tool but it gives information about program's execution during runtime. Tutnew is based on redefining 'new' and 'delete' functions of C++ language. Rational Purify [Rat00] is another tool for avoiding memory problems.

In addition to coverage analysis, it is possible to actually trace and collect events during runtime. Visualizing them is one way of dynamic analysis. Runtime events consist e.g. of system calls, X calls, object construction and destruction and exceptions. Investigating actual calls made at runtime gives a more detailed study of dynamic calls. Analyze of individual object lifelines can be dug out by investigating construction and destruction events. Object instantiations are an interesting thing to figure out. How many instances of one class are created during runtime? Do all objects behave similarly?

DYNO system introduced later deals with tracing individual calls and visualizing them.

2.3 Dynamic vs Static Models

The correspondence between static and dynamic models in an object-oriented language is depicted in figure 2.2. The dynamic behavior is concrete and the code itself is an abstract model of the system. Connection from abstract to concrete becomes clear when a program is executed. Actual objects are instances of classes. Between classes there are relationships that correspond to the actual interactions between objects during runtime. In a static model there are not so many relations between classes compared to dynamic trace of a program. One class in the static model might represent thousands of objects. Relations between classes in the static model possibly include quite a few dynamic calls.

Correspondences between static and dynamic models are quite understandable and straight forward on a low level of abstraction. In fact, concrete classes readable straight from the source code correspond to actual objects during runtime, and relationships between classes correspond to interaction between actual objects.

Models on higher level of abstraction are not so simple to dig out from static source code or from dynamic trace of a program. Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context [GHJV95]. Singleton pattern, for example, is a way to create code where there is possibility to create only one instance from one class during runtime by making the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created, and it can provide a way to access the instance. Creation and usage of this class does not produce any special use case or execution pattern. Execution pattern is a part of a trace of a program that is familiar from earlier behavior of the trace. Execution patterns are discussed more in Section 4.3.

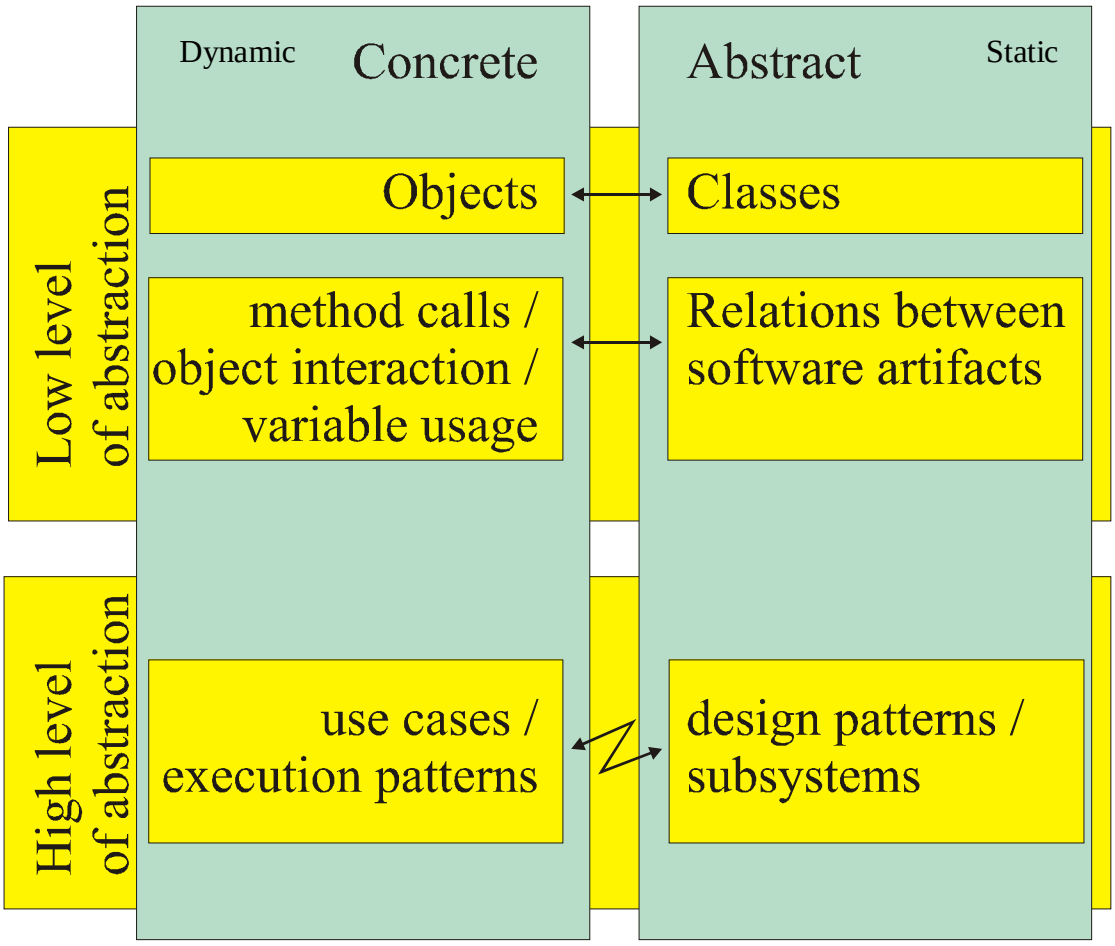

Figure 2.2: Dynamic vs static model

With one execution of a program produces one event trace. Event traces produced can be used to create alternative views to the static structure and dynamic behavior of the subject program. The event trace can be visualized, e.g., using UML sequence diagrams, which in turn can be transformed into class diagrams [SS00]. This way, a different view may be found to a system compared to the design of the system or the class diagram produced by static reverse engineering tools. By creating class diagrams from sequence diagrams, interesting ways to redesign systems architecture can be found.

3 UML Sequence Diagrams

Unified Modeling Language [Rat99c] describes a sequence diagram: “A sequence diagram presents an interaction, which is a set of messages between classifier roles within a collaboration to effect a desired operation or result.”

A sequence diagram shows an interaction arranged in a time sequence. In particular, it shows the instances participating in the interaction by their “lifelines” and the stimuli they exchange arranged in time sequence. A lifeline shows the existence of an object over a period of time. A sequence diagram does not show the associations among the objects like a class diagram. Between participants there are messages. Messages are presented in the top down order they are executed. Figure 3.1 presents a sequence diagram with concurrent objects.

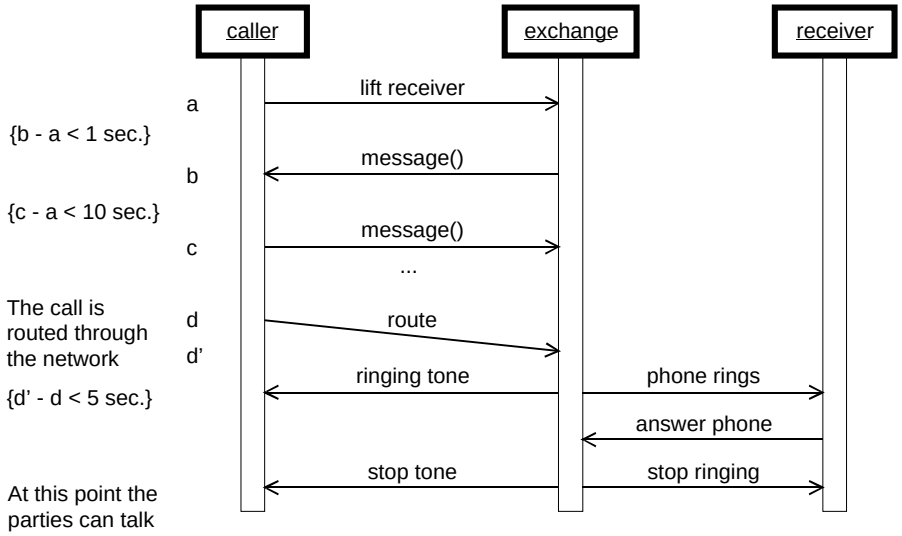

Figure 3.1: Simple sequence diagram with concurrent objects [Rat99c] (Notation guide)

Sequence diagrams appear in several slightly different formats intended for different purposes, like focusing on execution control, concurrency etc. A sequence diagram can exist in a generic form (describes all the possible sequences) and in an instance form

(describes one actual sequence consistent with the generic form). In cases without loops or branches, the two forms are isomorphic. Figure 3.4 is one example of a sequence diagram. It contains the following additional UML sequence diagram concepts: an object creation (e.g., op() creates the following object ob1), conditional branching (events [x>0] foo(x) and [x<0] bar(x)), conditional branches in the communication (branching dotted line of ob4:C4), a recursion (the object obj1 calls its own more() method), and an object deletion (crosses at the end of lifelines of ob1:C1 and ob2:C2). Branching shown as multiple arrows leaving a single point may represent conditionality or concurrency, depending on whether the guard conditions are mutually exclusive or not. The branching in figure 3.2 hence represents conditionality.

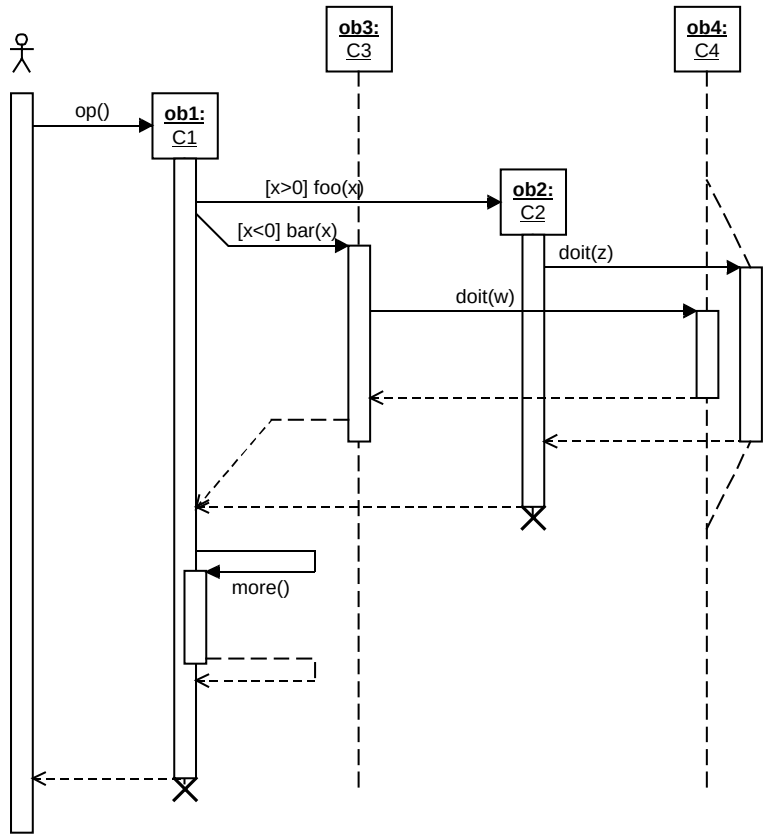

Figure 3.2: A sequence diagram with focus of control, conditional, recursion, creation, and destruction [Rat99c]

A sequence diagram is one of the behavioral diagram types in UML. Behavioral diagrams can be presented as collaboration in UML model. In figure 3.3, there is a simplification of collaboration model in UML.

An Interaction specifies the communication patterns between the roles. More precisely, it contains a set of partially ordered Messages, each specifying one communication, e.g. what Signal to be sent or what Operation to be invoked, as well as the roles to be played by the sender and the receiver, respectively.

It is possible to transform sequence diagrams into state diagrams easily [SKS00] because they are semantically close diagram types. The target diagram contains (almost) the same information as the source diagram.

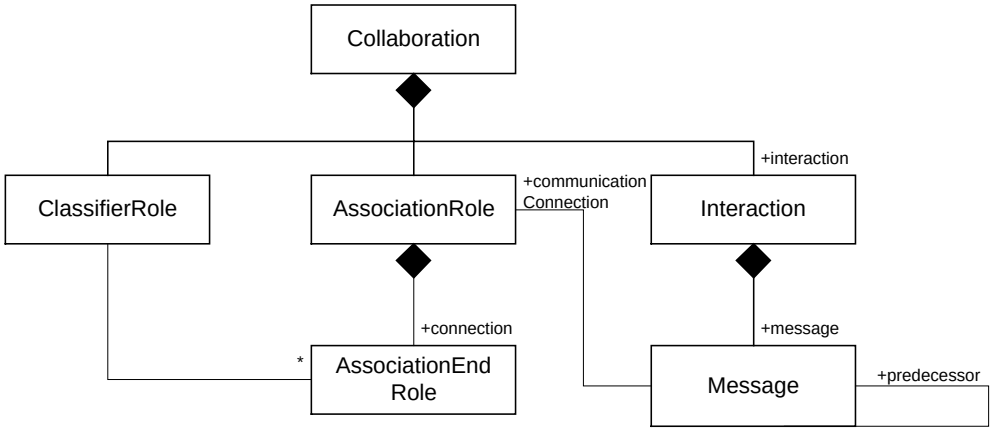

Figure 3.3: Simplification of collaboration model in UML

4 Visualization

4.1 Dynamic Visualization of Object-Oriented Systems

Most conventional object-oriented program analysis tools provide dynamic analysis in terms of lower level units like classes, instances and methods [DHKV93]. One example of such tools is the early versions of Program Explorer [LN95].

Because program execution produces massive amounts of information, most dynamic reverse engineering tools use some kinds of graph representations to visualize the interactions. In graph representation the figure processed is often quite close to a class diagram. It is not possible to see out the order of call sequence in a class diagram. However, it is possible to view the amount of interaction between objects, e.g. the number of calls made between nodes of a graph can be marked as a weight of an arc in a graph. This can be visualized with a simple number, by depth of coloring of the arc, or thickness of an arrow [SSC96].

Rigi reverse environment graphs are used for visualizing dynamic behavior of the program. Dali [KC98] system uses Rigi as a visualizer.

ISVis [Mor00] and Scene [KM96] are reverse engineering tools that produce sequential visualization of execution of a program. Both of them use different ways to reduce information that is visualized so that the user could deal with figures with human eye.

ISVis uses the static information of a program to choose those parts of a program in which the user is interested in. ISVis produces the whole scenario so that the chosen participants interact into the right part of window. By selecting a part from the larger trace it is possible to zoom a smaller part of a trace in the main window. By zooming more actual method names called are drawn in the view. Figure 4.1 shows the user interface of the ISVis tools.

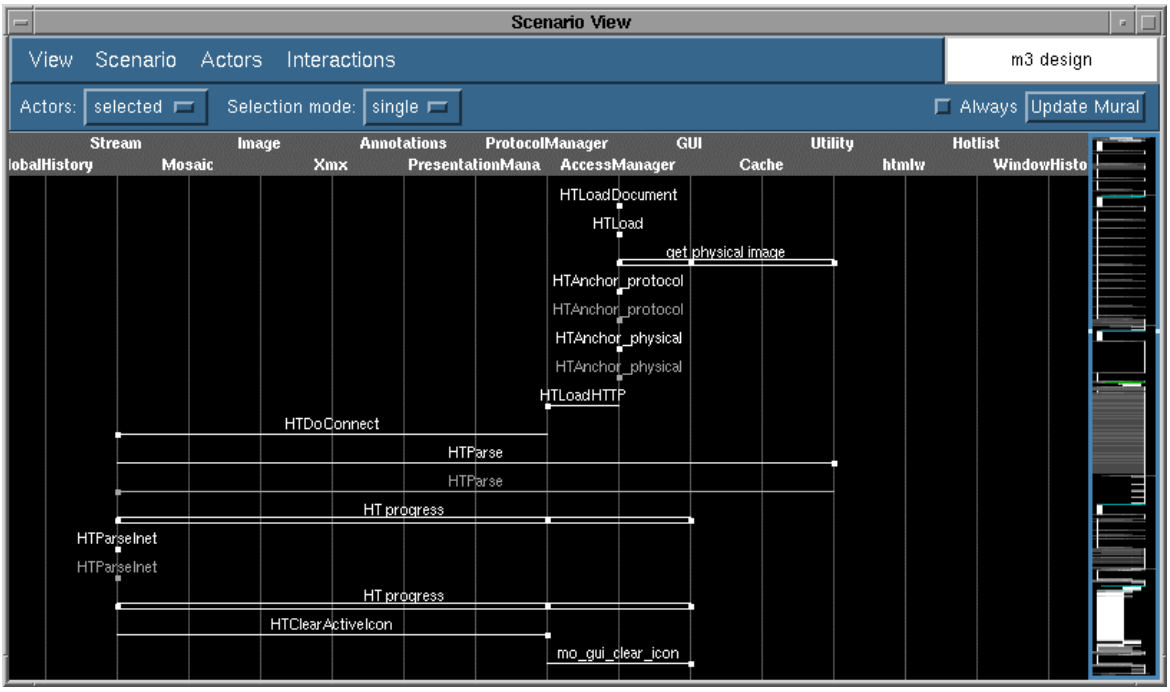

Figure 4.1: A scenario in the ISVis tool

Scene is also a reverse engineering tool that uses scenarios to visualize object-oriented program execution. It uses source code instrumentation technique to produce event traces. With Scene, it is possible to browse not only scenarios but several different kinds of documents like class diagrams, source code, class interfaces and call matrices. Scene also allows the user to investigate the inner state of an object in a specific point of execution. By default, Scene views a scenario in a closed form to reduce the amount of information that is visualized. Only those arcs are shown that are called from the highest level of a scenario. By clicking arcs more participants of a scenario are opened into the view. This way the user is able to dig inside a trace only so deep that he is interested in. In figure 4.2 there is a picture of a typical Scene diagram. In the Scene tool, a long trace is packaged in parts which can be opened. This way the length of a scenario is reduced. In figure 4.2 ‘Part 1’ on ‘Main’ lifeline is this kind of package.

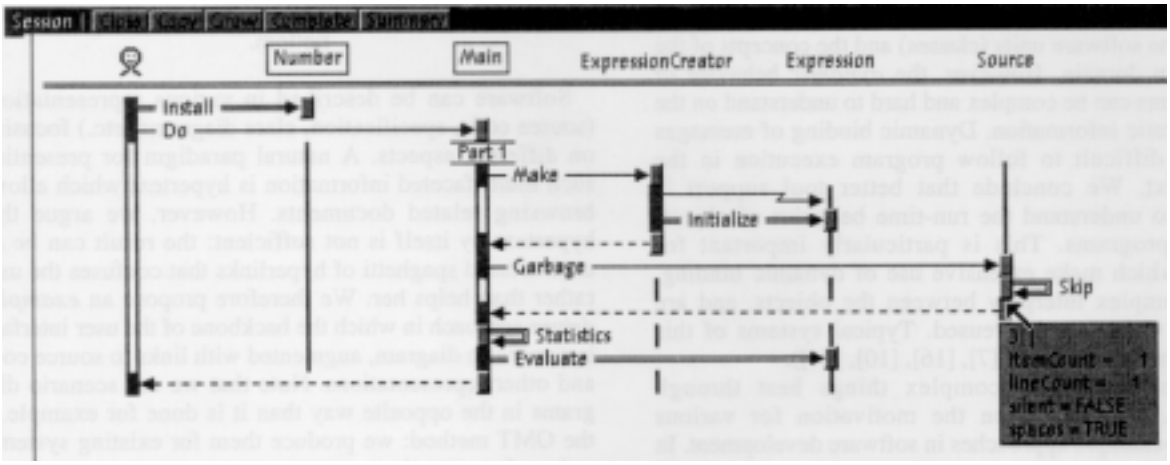

Figure 4.2: Scene diagram

4.2 Architecture-Oriented Visualization

In large software systems the following questions may arise: “How many times does this application process visit the file system?” or “Is this lock currently in use by any sub-framework of a virtual memory system?” In such cases, the behavior represented on class or object levels is not necessarily interesting.

Architecture-oriented visualization gives an opportunity to investigate the behavior of a program step by step. It allows the user to investigate a program from higher levels of abstraction. After finding the place in which there is a problem, it gives the opportunity create views in more detailed way. Moreover, it gives the opportunity to reduce the information to be presented in a visualizer.

Figure 4.3 gives a possible hierarchy of abstraction levels for architecture-oriented visualization. The highest level of abstraction could be considered as the system level, focusing on how do whole systems intercommunicate. Systems are composed of subsystems. Interaction between subsystems forms the next level. Frameworks are used to build systems and subsystems. Visualizing their behavior is the next abstraction level. Design patterns [GHJV95] are certain ways to implement code. The interaction produced by design patterns can be used as one abstraction level. Figure 4.4 shows the abstract interaction between classes that can be used to visualize design pattern behavior. A trace

of a program can be produced out of the interaction between actual classes. And in more detailed level, participants can be instances of classes – actual objects. Finally, it is possible to be interested in individual method calls.

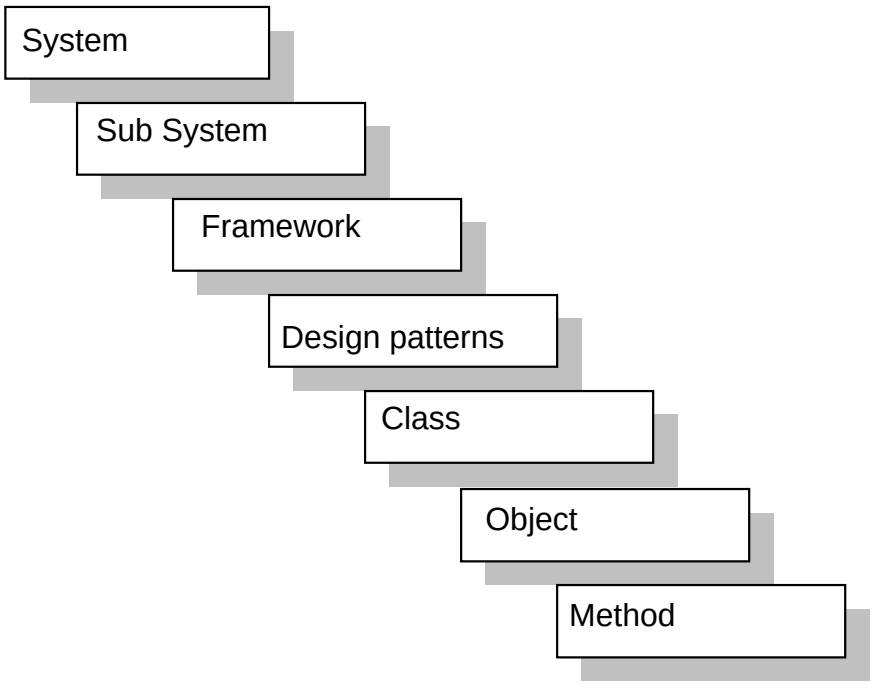

Figure 4.3: The levels of architecture-oriented visualization

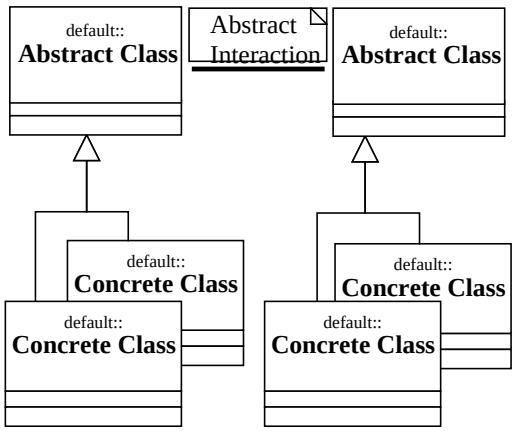

Figure 4.4: "Abstract Behavior" in Design Patterns [LN95]

To find the interesting parts of a program to be investigated, one should first try to get some kind of communication graph where the amount of interactions between parts of the program is shown in a higher level of abstraction. If there is a massive amount of interaction between parts where this should not occur, e.g. a performance bottleneck, we can go and investigate the interaction in more detail.

Collecting the trace between higher abstraction levels can be produced by architecture-oriented instrumentation [SSC96]. Another way is to package the calls made by classes and consider them as subsystems or parts of a framework.

4.3 Execution Patterns

Opening a dialog is an example of a behavioral pattern: the same sequence of events creating dialog items and giving them initial values is repeated every time the dialog is opened. Execution patterns are found by recognizing similar behavior in the earlier execution of a program [DLVW98].

Programming languages let us specify repeated executions of almost the same scenario through recursion, functional decomposition, template functions and variety of control structures. These quite compact ways to write code can produce long executions and traces that repeat almost the same pattern many times.

Execution patterns can be used in reducing the information that is visualized in a sequence diagram by collapsing a pattern as a subsequence. Execution pattern visualization tells a programmer how the program executes. When the information of how many times the execution pattern appears is measured, the bottlenecks in a program can be found.

Execution patterns can be recognized at many levels. The levels are quite similar to those used for architecture-oriented visualization. By generalization it is possible to loosen the demands of participants in the execution pattern.

The most restricting level of execution pattern identification is the identity level. The sequences that constitute recognized execution pattern have to be exactly the same on the object level. Exactly the same objects have participate the sequence. The messages between the participants have to be in the exactly same order and all the messages must appear in the sequences to be recognized as an execution pattern.

The first generalization level is the class identity level. There is no need that the participants of a pattern are exactly the same objects, it is enough that the participants are same in class level. In other words, the participating objects may differ but they have to be instances of the same classes.

Polymorphism is one way to generalize execution pattern recognition. It can be done considering classes as base class objects. For instance in a drawing tool one is not interested in which kind of object is drawn, a rectangle or a circle, but how the program runs while the object of any kind is drawn.

The next generalization levels are frameworks, subsystems and systems like in architecture-oriented visualization. Trying to find execution patterns on a higher level might give alternative ways to regroup architectural implementations.

Execution patterns can be found also if there is no interest of the participants of a sequence but only in the message structure in general. For example there might be a similar execution in creating a message box and a warning textbox.

Depth limiting is a way to generalize execution pattern recognition. Depth limiting means that a limit of calls is given, as to how deep into sequence the calls have to be similar. After that there can be any kind of behavior inside different patterns as long as the start of sequences are similar to the given depth.

In code, there are compact ways to produce quite long traces that repeat the same kind of execution many times. Recursion and loops produce this kind of execution. Identifying these repetitions is one way to find execution patterns.

Commutativity can generalize execution pattern matching. Commutativity means that A,B matches a pattern where the call order is B,A.

Execution patterns can be found one after another and one pattern can include several patterns. This is called associativity {{C,{D}},E} -> {C,D,E}.

Different generalizations can be used together. For example, using repetition and commutativity generalizations a freedom of the number of D executions above could be given, even though these executions could be considered as the same pattern. ‘CDDDE’ and ‘CED’ could be recognized as the same pattern.

Using tools for searching execution patterns supports execution understanding, since the execution trace can be visualized on a higher level of abstraction and the user can browse and learn the patterns.

To reduce the amount of information we might be interested only in ways how objects are created and destroyed, object lifelines. Then it is sufficient to show the trees which include creating and destroying.

How to find execution patterns? It is possible to try recognizing execution patterns during runtime if there is a library of patterns familiar to a programmer available. It is better to first collect the whole sequence and after that trying to find execution patterns from it.

Every action might be a start for a new pattern. One way to find patterns is to give a hash value to each node in a call tree. A recursive hash function computes a hash value for each subtree. The value depends on the generalization level. If several subtrees with the same hash value can be found there might be an execution pattern in place. One problem is that the hash value could be same to different kinds of patterns. To correct this problem one has to try to find a better hash function.

Considering a trace as a picture that is described in structural way and applying algorithms familiar from signal processing and image recognition might also be used to recognize execution patterns.

The level of pattern matching should also be chosen. Because every subsequence could be considered as an execution pattern, which execution patterns should be visualized as patterns? Should the user be asked or should it be automatically decided? Maybe both. The problem is how to choose the line between automatically identified and asked ones. One more possibility to tighten the execution pattern matching is by giving a treshold value for a pattern: a subsequence is considered as a pattern if it is repeated frequently enough.

When a program is analyzed with sequences, a trace is drawn out of every test dataset. Many traces for one program are hence produced. Similar patterns out of these traces can be found. For pattern identification, it might be useful to create a decision tree and use it to recognize patterns out of other traces of the same program. Updating the decision tree

with results of the other traces would be smart. Other machine learning methods might also be useful [RN97].

4.4 Visualization with TED

TED [Wik98] is a graphical software-engineering environment that supports UML. When creating UML elements into TED both model and view components of the elements have to be created before anything can be visualized. Model elements can be created alone. Both model and view elements can be investigated with Windows Explorer. The order of messages is not an implemented in TED model of UML sequence diagrams. This is quite vital in sequence diagrams if there is a need to read the sequences with other programs.

In addition to UML view TED presents specific view elements as workbooks. With workbooks it is possible to create a directory structure. Any UML diagrams can be drawn in workbooks.

In a TED view a sequence diagram can be drawn as shown in figure 4.5. Information text can be presented inside a UML note or a text field. Classifier roles can be drawn anywhere and the length of a lifeline can be modified. Marking classifier role to be destroyed by putting X at the end of lifeline is supported. A lifeline cannot be presented with a dashed line. Actions can be placed above the lifeline.

Messages can be attached between classifier roles. Messages can be slanted downwards or attached back to a caller as messages ‘msg’ and ‘msg2’ in figure 4.5. It is not possible to present creation of an object by drawing a line to a box that contains the name of a participant. A message can be attached only to a lifeline. The creation of a participant can be described in the label of a message. It is not possible to modify arrowheads. Thus, asynchronous messages cannot be described. The thickness and coloring of objects can be modified. The line type of message can also be modified. A return message can be visualized with a dotted line as defined in UML. The order of messages is only visual. Subscenario lines and condition lines between classifier roles are also available although they are not described in UML . The canvas size in TED is resizable, so large scenarios can be presented with it. Zooming is supported in TED. Large scenarios can be shown in one window and by zooming it is possible to see smaller parts of a scenario. Only one

instance of one workbook can be opened in one session. ISVis-like visualization in which there is a large sequence in other part [Mor00] in a window and a zoomed one in another part is not easily used in TED.

TED allows hyperlinking between any visual objects. For example a subscenario can be linked to a workbook as presented in figure 4.5. Created links work both ways. Several links can be attached to the same object.

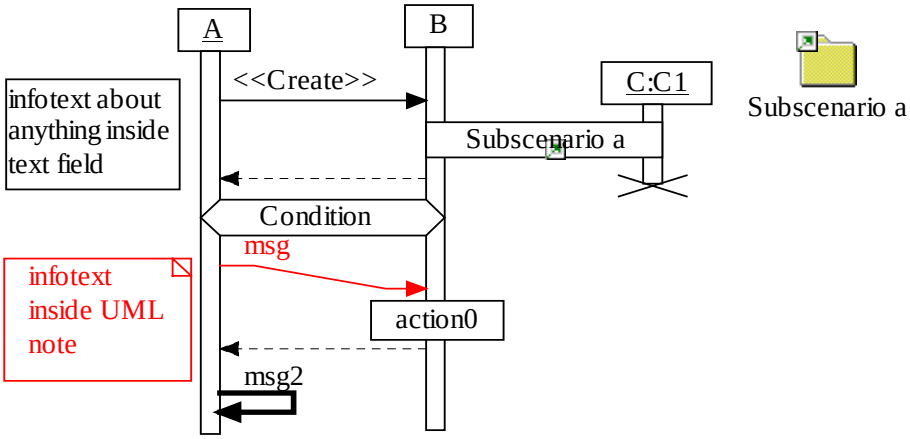

Figure 4.5: Hyperlinked subcenario in TED

5 Collecting Information for Sequence Diagrams

During the runtime of an object-oriented program, information should be generated and collected in order to produce sequence diagrams. The program in question should

1) generate events for object construction and destruction
2) generate events for method entry and exit
3) collect static type information, such as class structure and member declarations
4) collect dynamic type information to resolve an instance's class at runtime

In addition there could be debugger-like additions in a reverse engineering tool. Support for suspending and resuming subject system execution is one thing that might be important. When the execution is paused the ability to examine and explore a subject's internal state allows the user to investigate runtime values and the behavior of the code. In Scene [KM96] also the values of variables are collected in different phases of program execution.

5.1 Instrumentation

Source code instrumentation is the most commonly used way to produce runtime information. Instrumentation adds new objects into an existing code. These objects are used to collect events.

Instrumentation can be produced by hand. Manual instrumentation is time consuming and the results would probably be erroneous, so instrumentation tools are used. Instrumentation tools are based on parsers, which are used to collect static information out of code. Instrumentation tools use parsers to find out places in code where to put instrumentation additions.

If the grammar of a computer language is simple the results of a good parser are accurate. For instance languages that use LL(k)- or LR(k)-grammar can be parsed efficiently. Since C++ is quite a versatile computer language, different C++ parsers usually produce slightly

different static models out of identical C++ code. Therefore the instrumentation of C++ code should be produced in those parts of a code that are easily detected.

The user has to be certain that the code under inspection is of the same version that the subject system has been compiled from. The unavailability of source code makes the source code instrumentation technique useless.

5.2 Alternative Ways to Collect Runtime Events

It is possible to produce event traces by compiler modification. Program Explorer [LN95][LN97] has moved in its later version to using a debugger-like solution to solve problems in understanding the static structure of a program when the source code is unavailable. This solution together with instrumentation techniques makes it possible to reduce the amount of information that is collected.

Tarja Systä presents a way to collect information of Java programs in her doctoral thesis [Sys00]. She uses a technique to read static information straight from the byte code and modified JDK debugger produces the events. This way the user can be certain that the subject system is the same as the used version. There is no need to modify and recompile code.

Jinsight tool is a tool for visualizing the dynamic behavior of Java programs [IBM00]. In Jinsight tool the event traces are produced as a result of instrumenting Java Virtual Machine.

5.3 Objects in C++

Object-oriented programs are rich in structure: methods and attributes belong to classes, objects are instances of classes, and interclass relationships entail association, aggregation, inheritance and call relationships. Struct and union are also types of class in C++, so instances of them could also be considered as objects.

Because of dynamic binding it is often difficult to know which part of the code is actually executed. Dynamic binding occurs when an inherited class is combined from base classes during compiling. Functions can be reimplemented in inherited classes. If there are several

levels of inheritance it might be difficult to notice which implementation of the function is executed. In figure 5.1 one can see the idea where different parts of code are executed when an object is called. For example, there might be a function implemented in C1 and C2. During execution, an instance of C3 is called with this function and code written in class C2 is executed.

In figure 5.2 a code example is given where fun( ) written in class C1 is not used in objects type of C2 or C3. Instead fun( ) written in class C2 is used at runtime.

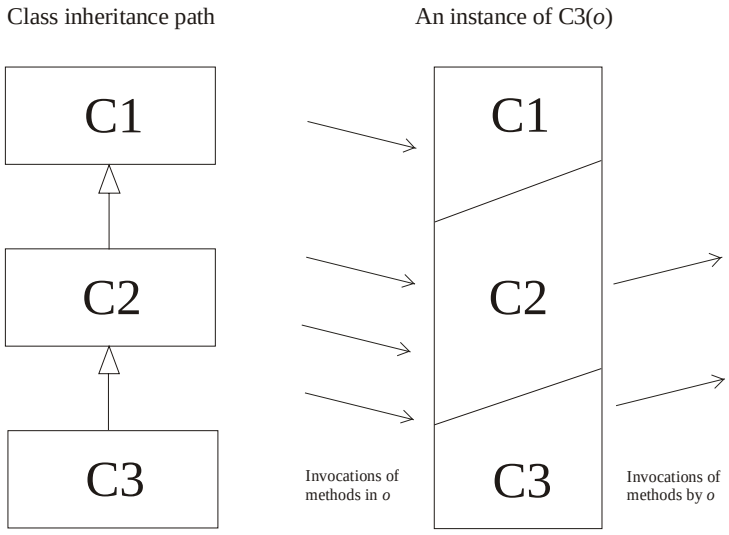

Figure 5.1: An object of inheriting class [LN97]

```
class C1 {
      public: int fun(int i) { i++; return i; } };

class C2 : public C1 {
      public: int fun(int i){ return i; } };

class C3 : public C2 {};

int main(){
      C3 op;
      int i;
      i = op.fun(4);
      return 0; }
```

Figure 5.2: Sample code of function fun( ) that is overriden by dynamic binding in class C2

In C++, it is possible to track the usage of functions by adding an object in the beginning of a function. When the added object is created it sends information about the function and object, which is the receiver of the message. When the function ends, all local objects are destroyed in C++. Also the added object is destroyed and it sends information about returning from called object back to the caller. That is why the sequence is produced like in figure 5.3 and sequences like in figure 5.4 are not possible.

Because there are many ways to call functions in C++, it is difficult to track all function callers. So this way only the function call's arrow heads and the return calls are collected. To find out who is the caller of the function is it is necessary to collect all messages from program execution. In the beginning, there is a main function after which it is possible to track the callers when we know the whole sequence.

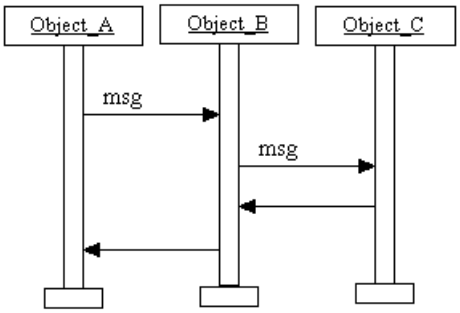

Figure 5.3: Simple sequence

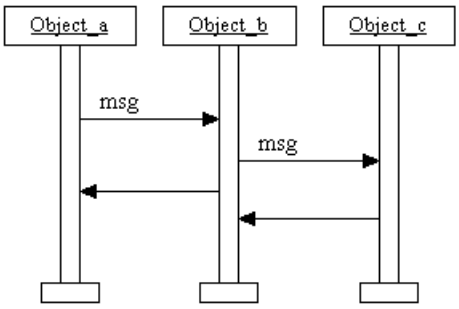

Figure 5.4: Impossible sequence in object-oriented program

The object's lifeline is one interesting thing in dynamic analysis. In C++, it is possible to write a class without constructor and/or destructor functions. The creation or destruction of an object is missed when only adding observer objects into functions. This can be avoided by adding a base class in all classes. This base class works similarly to the object added into functions. It notifies about messages when it is created and destroyed. The information of which kind of object is created can be found out by making this base class a template base class with template type of the inherited class.

Only the message receivers are collected by added object creation. When the added object is destroyed, it is a target of the return message to the object, which called the function.

5.4 Notation of a sequence [LN97]

There is no sequential conditionality or branching included in one trace of execution of a program. Form of collected information can be described with a notation. Concurrency is supported in UML sequence diagrams. If there is concurrency, this notation describes one single thread only.

Notation $o^{c,d}$ can be used for objects and $(o_1,o_2)^{e,r}{}_m$ for arcs. The set of objects o_i created during execution is O. O consists all $o^{c,d}$ where c and d (creation and deletion) denotes objects lifetime; c and d should be non-negative and c should be less than d. Within O no two objects should have the same time of construction or the same time of destruction.

During its lifetime an object can invoke methods of other objects and/or itself. Object interaction can be modeled as an interaction graph G_i = (O, A, ei), where A is a binary relation on O (also called a set of arcs) ei is the entry invocation.

The notation $(o_1,o_2)^{e,r}{}_m$ is a labeled arc in A, where m denotes the method invoked in o_2 and where e, r is the activation. Here, e denotes the time at which this invocation takes place, and r denotes the time at which the execution returns to o_1. Both e and r should be non-negative, e should be less than r, and each should be within the lifetime of both o_1 and o_2. The entry invocation is a special arc $(\_,o)^{e,r}{}_m$

Some method invocations do not result in any further invocations. These are called terminal invocations. An activation path is a sequence of arcs $< a_1, a_2, \ldots a_n>$, where the first element a_1 is the entry invocation, the last a_n is a terminal invocation, and each a_{i+1} is

with in the activation (e, r) of the previous a_i. Examples of activation paths in figure 5.5 are $< a_1, a_2 >$, $< a_1, a_3, a_4 >$ and $<a_1, a_5>$. a_{i+1} is a subinvocation of both the invocation a_i and the object o_y. A partial activation path is any sublist of a full activation path.

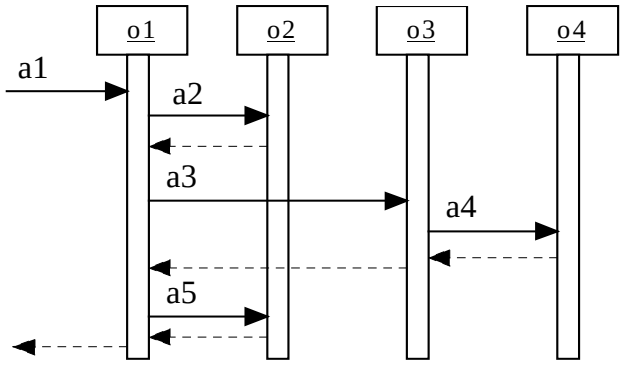

Figure 5.5: A sequence diagram

6 DYNO

In this chapter DYNO system is introduced. It is a system for producing information needed to draw sequence diagrams. Ways to reduce the amount of information to be collected are introduced and methods of architectural visualization are used. After execution of a subject program there exists a trace of the execution, which is used to create a sequence diagram in TED design environment. Execution pattern identification options may be used when importing a sequence to a visualizer.

6.1 Overview to DYNO

DYNO collects runtime data of a subject non-concurrent C++ program. Information about classes and methods has to be collected and certain objects must be added into C++ code to collect the data about the dynamic behavior of a program. The structure of DYNO is described in figure 6.1. With an instrumentation tool called caninstr an .ifo file and an instrumented C++ file are produced based on the original C++ file. Instrumentation adds “CDD” objects into the code. These are the link between the actual program and DYNO. The instrumented file is compiled. Before running packaging and filtering options are selected in the user interface of DYNO to reduce the amount of information to be collected. Prefilter collects the events from “CDD” and modifies information in such a way that a sequence diagram can be drawn after it. Unnecessary information is filtered out and the packaging can be performed. A component called classbase, used by other components of DYNO, includes the static information of a program described in the .ifo file.

After the execution of a program a .dat file is created. It includes the information to be imported into a sequence diagram. TED is used as a visualizer. Tedseqimport is a component that is used for reading the .dat file sequence into TED.

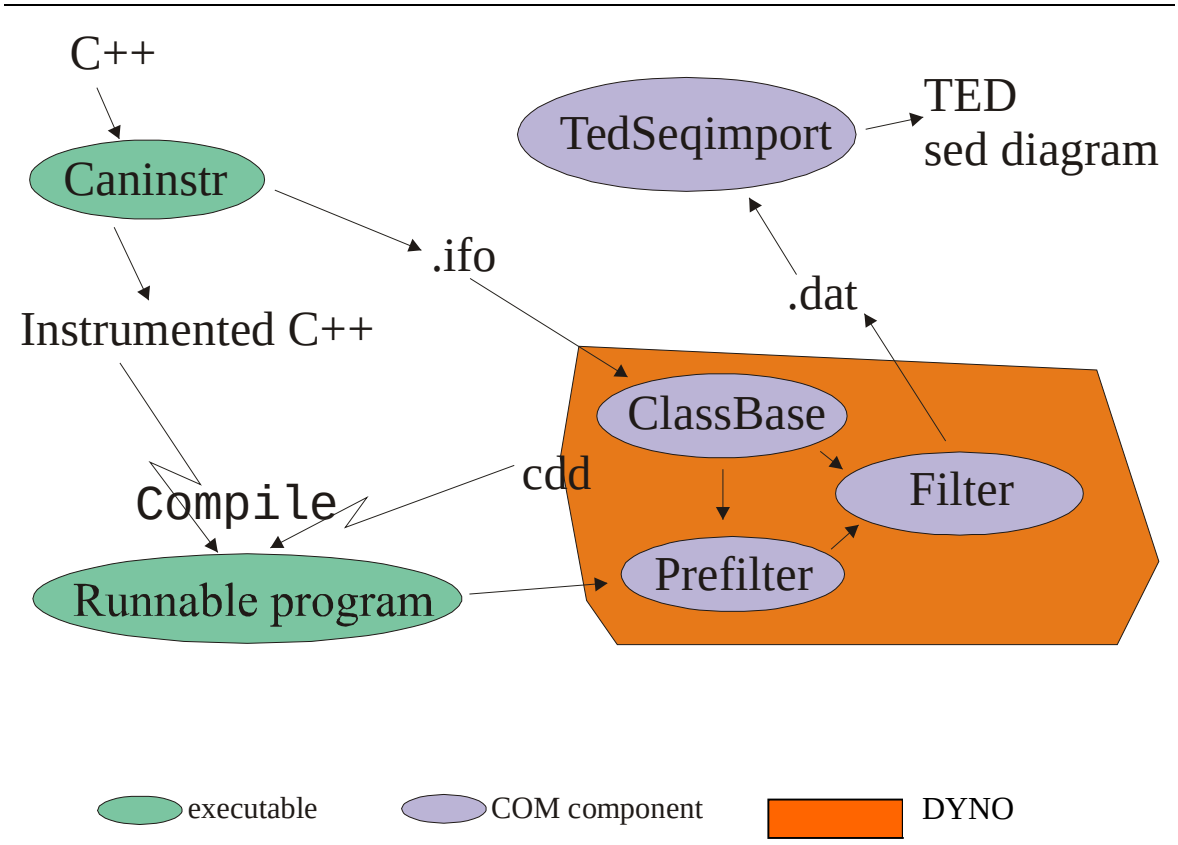

Figure 6.1: DYNO

6.2 Instrumentation

Caninstr makes the instrumentation. Program caninstr is produced and based on the CAN project's [Nok99] C++ Analyzer. CAN is a project for producing a special reverse engineering tool for Nokia's software designers. The instrumentor is produced in MTA-JATE Research Group on Artificial Intelligence (RGAI) in Hungarian Academy of Sciences -University of Szeged.

Caninstr reads the file to be instrumented and writes the resulting instrumented file into a subdirectory called "instrumented" with the original filename. The static information of the program is collected into an .ifo file that is put also in the same subdirectory. The contents of the .ifo file produced from code in figure 5.2 is shown in figure 6.2. Class and method names are collected, and given a unique identification number. Class hierarchy is described in the end of the .ifo file. The additions that caninstr instruments into a program code are described in figure 6.3.

```
#Instrumentation information for file ‘overfun.cpp’.
#Generated on Mon Nov 13 19:17:52 2000.
#Class/function ids:
[CLFCID]
1;C1;C
2;fun;F
3;C2;C
4;fun;F
5;C3;C
6;main;F
#Class inheritance:
[CLINH]
3;1
5;3
```

Figure 6.2: An example of an .ifo file produced by caninstr

1. “**`#include __cdd.h`**“ into include section of the file.
2. base class __cdd<> to definition of every class. For example

   a) `class X {`

   becomes -> `class X : public __cdd <X> {`

   b) `class Y : public X {`

   becomes-> `class Y : public X , public __cdd <Y> {`
3. dyno object into beginning of every method

   `__cddfun __cddob (typeid(*this),num);`

   where ”num” is the identification number of the method described in .ifo. file
4. **`__cddinit`** object into beginning of main section.

Figure 6.3: Instrumentation additions by caninstr

Caninstr handles only one source file at a time. If a project consists of several C++ files, the project has to be preprosessed to .p form in a compiler before instrumentation. Preprosessing of a program means that all separate code files of a project are collected inside a .p file including library files. The user is not usually interested in functions described in library files when drawing a sequence. So unnecessary functions are instrumented. That info has to be filtered out afterwards and these unwanted events slows down the execution of the instrumented program.

6.3 Event creator CDD

CDD is the link between the code of the subject system and DYNO. CDD objects inform DYNO about the runtime events during the execution of a program. CDD objects *__cddinit*, *__cdd* and *__cddfun* are instrumented into C++ code to be analysed. They are inherited from class *senderi* as described in figure 6.4. Creation and destruction of objects in CDD all produce different kinds of events that have to be sent forward. *Senderi* handles the sending. *Senderi* is selected to be a base class of CDD classes because all CDD classes send events forward.

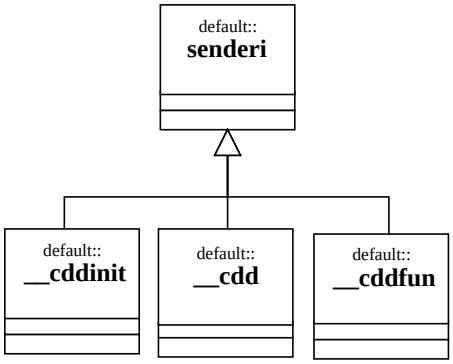

Figure 6.4: CDD

__cddinit wakes up DYNO components and sends the first message of a sequence; this is the beginning signal. *__cddinit* is used only once in the beginning and once in the end of a program. In the beginning DYNO components classbase, prefilter and filter are started. When a program ends *__cddinit* object is destroyed, and it sends the end of program message forward and closes DYNO components.

When *__cdd* or *__cddfun* object is created or destroyed *senderi* sends information of this action to the prefilter.

Senderi gives all messages an identification number that can be considered the order number of the message. The same number can be used as c, d, e and r in the notification described in the end of Chapter 5.

Information of one message that *senderi* sends includes

- The order number of action
- The type of action

  0 – initialization from __*cddinit*

  1 – create object from __*cdd* creator

  2 – destroy object from __*cdd* destructor

  3 – start of method from __*cddfun* creator

  4 – end of method from __*cddfun* destructor
- The identification number of a method when type of an action is “3 start of method”
- The number of object (value of *this* function) from __*cdd* or id number of a method from __*cddfun*
- The name of the class

6.4 Prefilter

Prefilter receives information of an action from CDD. It

1) filters the base class creations and destructions actions away from inherited classes,
2) adds the caller information to every message,
3) filters the unnecessary end of method messages and return messages away and
4) filters out unnecessary destruction messages.

Instrumented code introduced in figure 6.5 produces events described in figure 6.6. Instrumented parts of the code are printed in *italics*.

```
1     #include __cdd.h
2     class C1 : public __cdd<C1 > {
3     public:
4           int fun() { __cddfun __cddop(typeid(this),2); return 1; }
5     };
6
7     class C2 : public __cdd<C2 >{
8     public:
9           C2( C1 c ){ __cddfun __cddop(typeid(this),4);
10                i = c.fun(); }
11    private:
12          int i;
13    };
14
15    class C3 : public C2 , public __cdd<C3 > {
16    public:
17          C3( C1 d ) : C2( d ) { __cddfun __cddop(typeid(this),6); }
18    };
19
20    int main(){__cddinit __cddop;
21          C1 a;
22          C3 b( a );
23          return 0;
24    }
```

Figure 6.5: Example code

The prefilter memorizes *this*-value of the object when an object is created. When an object is destroyed, the identification number of the object inside prefilter is destroyed. The same *this*-value can exist for one object at a time. Only those destruction events are interesting that have correponding creation event. These creation events are memorized by prefilter. In figure 6.6 destruction events of class C1 ‘8’ and ‘12’ are unwanted for constructing a sequence.

As can be seen in the main function, no instance of C2 is created. Actions ‘3’, ‘4’, ‘7’ and ‘14’ coming from class C2 are actually part of creation of an instance of class C3. The static information about inheritance is needed to figure out which instance is being created. The actual sequence after prefiltering is shown in figure 6.7.

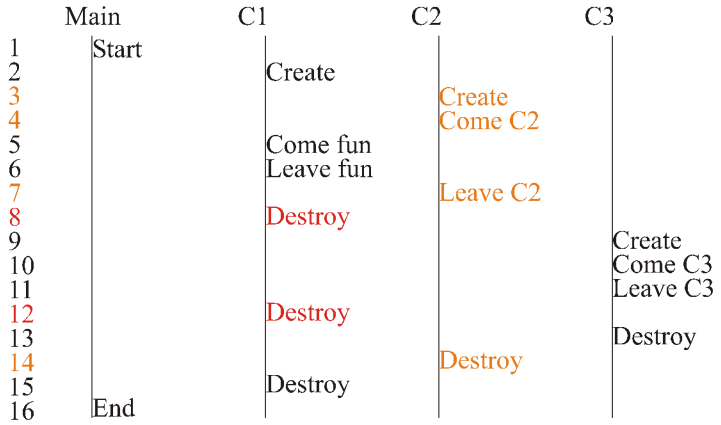

Figure 6.6: Events collected by prefilter

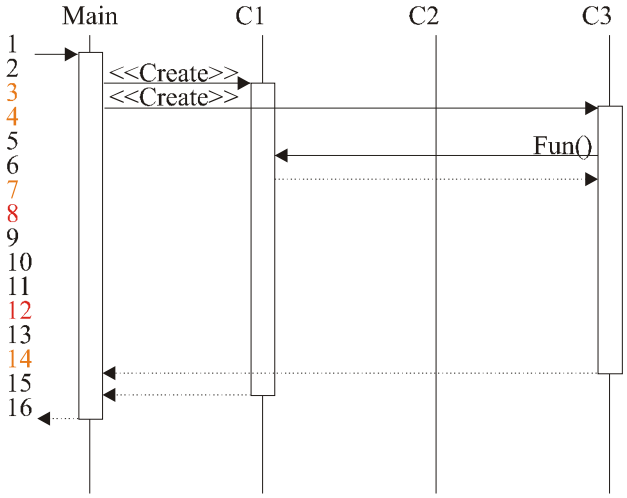

Figure 6.7: Sequence after prefiltering

Figure 6.8 illustrates the states of the prefilter when an object is created. During creation process all events are collected to a *maybe* list. When the type of the class can be identified, there is no possibility for it to become an instance of any other class, so the *maybe* list is changed to be sure. There might be other objects created inside creator function. Those creation events invoke a new maybe list. The state machine in figure 6.8 could be said to be a recursive state machine. When there are other objects created inside creator there becomes a new recursion level.

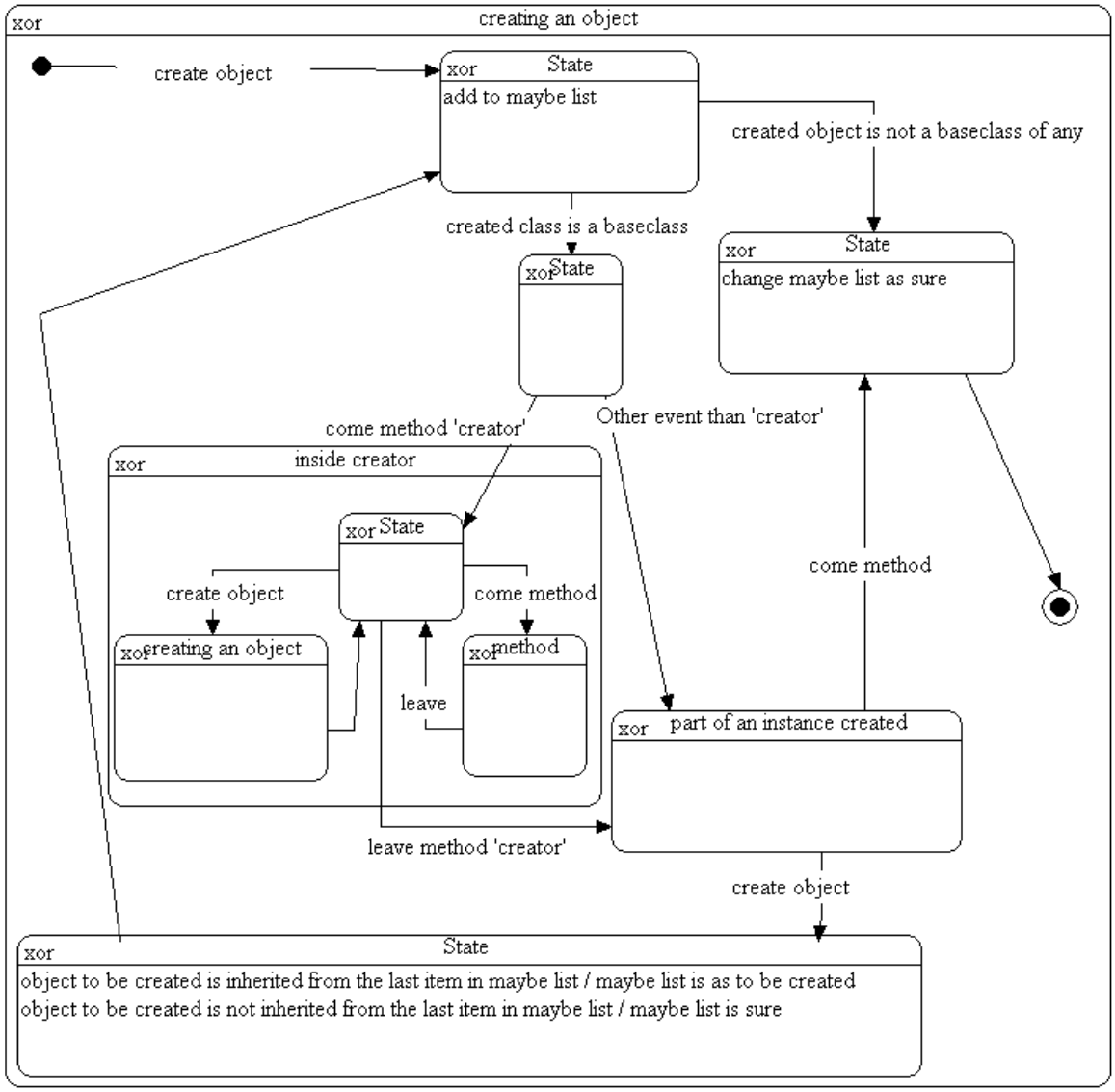

Figure 6.8: States of the prefilter when object is created

After the creation process destruction events are not so difficult to figure out, because *this*-values are memorized by prefilter. In addition to 'this'-values, there is stored which *this*-values represents actual instances and which are parts of other instances. When an actual instance is being destroyed there comes first information from the actual class like the event '13' from class 'C3' in the example. After that all destruction messages from base classes should appear. The difference compared to creation process is that the prefilter knows which kind of an instance is being destroyed, and messages can be drawn instantly. No *maybe* list is required in this case. In object-oriented programs, destruction of an object

usually leads to destruction of the objects that it has created. After prefilter there is information of the whole sequence to be sent to the filter. All calls are synchronized and they all have an existing return call. Notation G_i = (O, A, ei) described in the end of Chapter 5 can be used for a sequential event trace after prefilter.

6.5 Filter

Filter gets the whole sequence from the prefilter and sends further only those actions that are to be visualized. The filter packages and dismisses information according to selections of the user. The user interface component uploads the static information of the program and illustrates it to the user in a tree structure from which the user can choose the ways the information is filtered. These selections are made before the subject system is actually running.

Figure 6.9 shows the user interface of the filter. On the left side there is a “real classes” window, where all classes in the subject system are shown in a tree structure. On the right side there is the “class groupings” window. In this window methods of architectural oriented packaging are operated by the user. The next level under “virtual root” node contains the actual participants of the sequence that is produced after filtering. New groups can be added to present subsystems or groups as “My Group” in figure. Unwanted classes can be ignored. The hierarchy can be deeper than one level but all classes under the first level node are considered as the first level node. For instance an interaction between “Tuplaattori” and “Levitaattori” is considered as a call from “Esine” to “Esine”. In figure 6.10 the class hierarchy of figure 6.9 can bee seen.

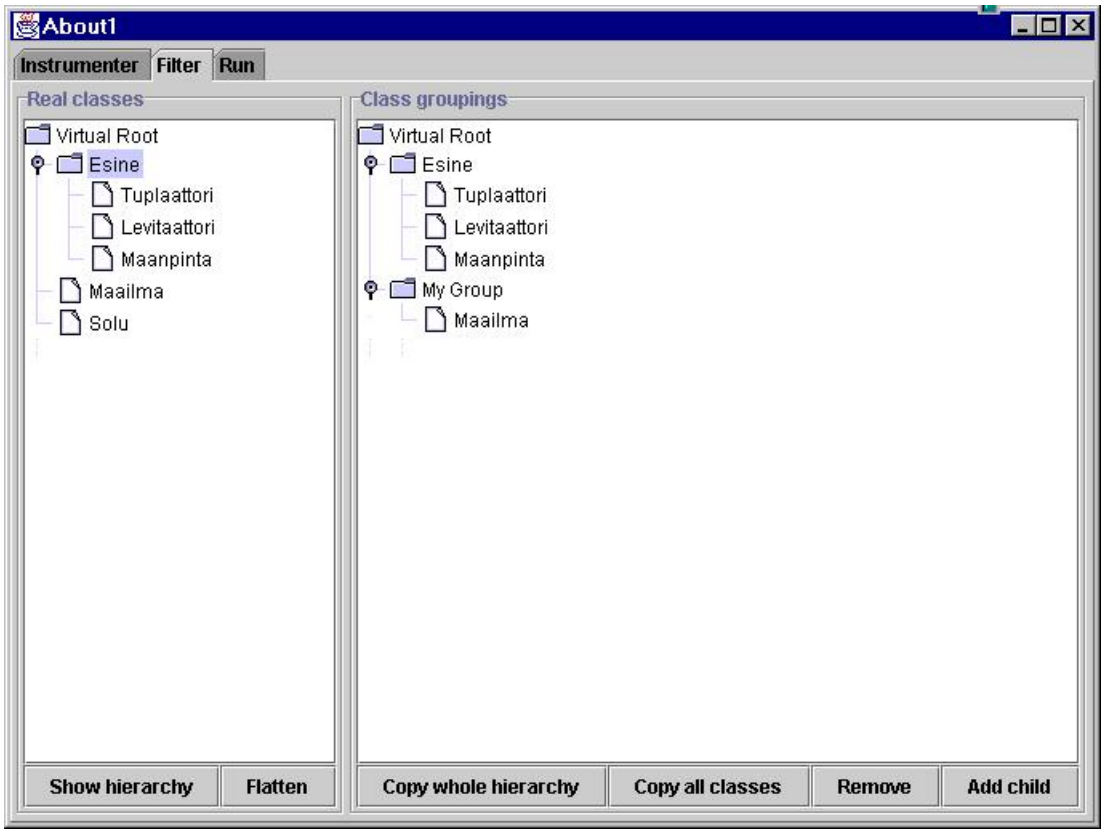

Figure 6.9: Selecting and organizing classes to be visualized

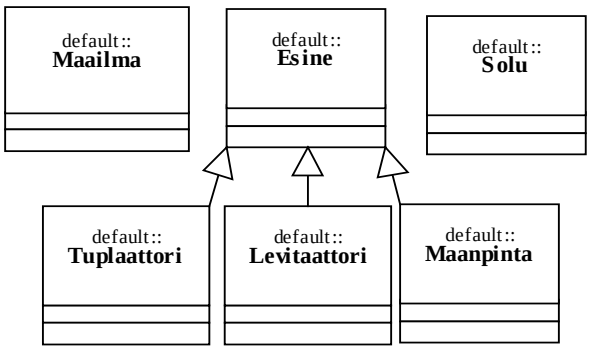

Figure 6.10: Class hierarchy of a program analyzed by DYNO shown in figure 6.3

Classes in the subject system are presented in the user interface in a form of inheritance hierarchy by default. Figure 6.11 shows the abstract-like interaction that would be collected when the inheritance hierarchy is used as the base of grouping. The other default way to present classes is “as they are” all on the top level by using “flatten” command. No packaging is used with this command.

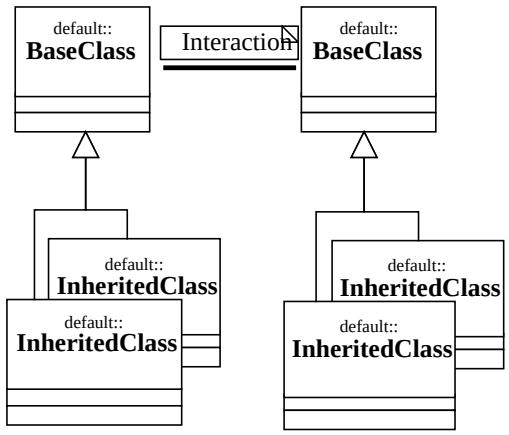

Figure 6.11: Default interaction to be collected with DYNO after filter

After selecting the classes and packages to be monitored the subject program is run. If the user is only interested in one part of the execution time of the program, filtering can be used. The user can select the amount of messages to be collected from “run” page of the user interface of the filter. By pressing F11 and F12 the user can select moments when the collection of events is started and stopped, if the option is selected from run page. By letting the user select the moment when the collection of a sequence information starts, the sequence entry point would probably be deeper in the sequence. The activation path, which led to the specific point, should be included to the produced sequence.

The collected data is stored in .dat file. The place and name of the .dat file is set on the run page**.**

If no time filtering is used, .dat file describes the whole sequence of a program after the execution of a program. The form of .dat file is described in figure 6.12 in Backus-Naur Form (BNF). “Trace” is the whole sequence. “Start” is the initialization message. For

every method entry “come” there is a “leave”. As additional information to this, BNF is known for having as many Create calls as Destroy calls.

```
Trace       ::=   Start A End
A           ::=   ε | Create (B)* Destroy
B           ::=   Create | Destroy | Method
Method      ::=   Come (B)* Leave
```

Figure 6.12: The BNF of a whole trace in .dat file

6.6 Importing a sequence to TED

When filtering, the architecture-oriented reduction of information has been used. After the restrictions are given to the filter and the program has been run, no additional information can be included in the event sequence afterwards. Filtering is implemented for cutting out interaction containing unwanted parts of a program and for reducing the amount of information to be collected. A similar packaging system as in the filter could be used when importing the produced information to a visualizer. Before importing actual sequences into the visualizer, execution pattern identification algorithms could be used. Execution pattern identification might be implemented here.

There is an option for filtering out all activation paths that do not include an object creation or an object destruction. The user might be interested only in the creation and destruction events. In that case, no other messages than creation and destruction are visualized.

6.7 Performance Test

It is possible to create sequence visualization online without temporary buffer between the filter and tedseqimport components, but it slows down the performance of original program dramatically. That is the reason why the results are first written into a file and then imported into TED, after the execution is completed.

We tried a small test program that had six classes with 450 MHz Pentium II, 265Mb memory, Windows NT 4.0 with service pack 5 and TED 1.0 service pack 3.

The execution time of the original program was instant with execution producing about 7000 messages. The instrumented version took about 3 seconds to complete and produced sequence into a text file size of 176 k. Importing that text file into TED repository took about 50 minutes to complete. The repository size increased 27 M from 21M to 49M. After rebooting the computer TED client could not open the workbook in which the sequence was written in. It can be therefore seen that the actual visualization of the sequences with large size is not reasonable in TED. In Windows Explorer from model side the interaction in which the result was opened in couple minutes. The messages were inside the TED repository in a random order.

6.8 Discussion

Running the instrumented program is slower than running the original one. Writing sequences into a file takes some time if the amount of events increases much like seen in the performance test. A buffer component will be needed; one that receives the messages and writes the produced events into a file in bigger blocks than one at a time.

TED is not usable for visualizing large scenarios in one view. A limit of events that are put into one single view –workbook must be set. It must be able to import lists into a repository.

50 minutes to import only 7000 messages is way too inefficient. Another slider program [LN95], with 100 lines, and only one window with scrollbars on bottom and on the right side, uses about 100 classes and a typical execution creates more than 3000 objects and twenty thousand events.

Finding out subsequences that are executed often and printing them only once into own workbook will not solve problems in time consumption. Moreover, it does not avoid the explosion of the size of the repository in TED can not be avoided either.

7 Conclusions and Future Development

In this thesis an overview to reverse engineering and especially dynamic reverse engineering of object-oriented software systems was given. A DYNO system was introduced. DYNO collects information about runtime object interactions. Produced sequences are visualized in TED software-engineering environment.

Suggestions to enhance instrumentation are support more than one source file and Slicing before instrumentation. When instrumentation of more than one sourcefile is supported it is unnecessary to produce precompiled .p files, which include unwanted library classes. Using slicing the instrumentation is done only to those parts of a program which the user is interested in. The amount of unwanted messages is reduced and the instrumented version of the program will not be as slow as with the complete instrumentation.

The exceptions are not handled by DYNO. Try- and catch blocks are not instrumented in DYNO. When an exception is thrown in the C++ program the execution is moved to the exception handler. Objects handled inside the block where the exception was thrown in are destroyed. If there is no handler of thrown exception in the next catch block the exception handler moves backwards in the programs execution and tries to find a handler to the exception from the earlier phases of program. DYNO can not produce information from the executed exception and therefore loses track of the position where the program code is been presently executed. Ways to produce information that includes exception handling should be investigated and implemented into DYNO.

Lightening of the repository used by TED has to be done before sequences from real systems can be visualized. Importation abilities of new artifacts to TED should be speeded up. The performance test clearly shows the need for this. Another solution could be to use another visualization tool. The use of COM components makes it quite easy to create importation components to other visualizers. In addition, on-line importation of subsequences could be considered. Only those parts of sequence diagrams would be imported that the user is interested in.

Finding subsequences should be implemented to be able to visualize them. Hash function should be selected and implemented. The use of other techniques to find subsequences should be considered.

In TED, subsequences can be drawn into different workbooks. These can be then linked to subscenario figures in actual parent scenario. A subsequence describes the transaction in more detailed way. There can be many levels of subsequences. The same subsequence could appear several times in the same sequence. Deciding how many subsequence levels are allowed, and how complex a sequence should be before it is separated from a main scenario, are things to be investigated.

Expanding ability of DYNO to handle concurrent systems would need identifying separate threads and collecting sequential information for all separate threads at the same time. Problems in synchronizing should be considered if concurrency is included in DYNO.

COM components could be considered as participants of a sequence. Support for interaction between COM components or other interoperable components would be one interesting point to investigate. Problem in COM components will be the unavailability of the source code, because there are often third party components in a system and the user does not know anything else about the component than the interface.

DYNO is useful when the sequential information is used to create other UML diagrams out of the produced trace when no sequential visualization is produced out of it. [SKS00] [SS00]

DYNO could be used to collect different traces of execution of a program with different datasets. These traces could be stored, and by selecting classes from a class diagram and maybe by introducing slicing techniques several different sequence diagrams could be visualized to give the user a view of how instances of these classes interact.

8 References

[Ban98] Bansiya J., *Automating Design-Pattern Identification, Dr. Dobb's Journal June 1998*, http://www.ddj.com/ 2000.

[Test00] Testwell Oy, *CTC++*, http://www.testwell.fi/, 2000.

[DHKV93] De Pauw Wim, Helm Richard, Kimmelman Doug, Vilssides John, Visualizing the Behavior of Object-Oriented Systems, In *Proceedings Of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA –93)*, ACM Press, pp. 326-337.

[DLVW98] De Paw Wim, Lorenz David, Vilissides John, Wegman Mark, Execution Patterns in Object-Oriented Visualization, *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS –98)*, Santa Fe, New Mexico, April, 1998, pp.219-234.

[GHJV95] Gamma E., Helm R., Johnson R., and Vlissides J., *Design Patterns: Elements of Object-Oriented Software Architecture*, Addison-Wesley, 1995.

[IBM00] IBM Research, *Jinsight, visualizing the execution of Java programs*, http://www.research.ibm.com/jinsight/, 2000.

[KC98] Kazman Rick, Carriere Jeromy, View Extraction and View Fusion in Architectural Understanding, *Proceedings of the Fifth International Conference on Software Reuse (ICSR5)*, 1998.

[KM96] Koskimies Kai, Mössenböck Hanspeter. Scene: Using Scenario Diagrams and Active Test for Illustrating Object-Oriented Programs, *Proceedings of the 18th International Conference on Software Engineering (ICSE –96)*, ACM Press, 1996, pp.366-375.

[LN95] Lange Danny B., Nakamura Yuichi, Interactive Visualization of Design Patterns Can Help in Framework Understanding, In *Proceedings Of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA –95)*, ACM Press, 1995, pp. 342-357.

[LN97] Lange Danny B., Nakamura Yuichi, *Object-Oriented Program Tracing and Visualization, IEEE Computer*, 30, 5, 1997, pp. 63-70.

[Mor00] MORALE project, *ISVis tool*, http://www.cc.gatech.edu/morale/tools/, College of Computing Georgia Institute of Technology, 2000

[Mül00] Müller Hausi A., *Understanding Software Systems Using Reverse Engineering Technologies Research and Practice*, http://www.rigi.csc.uvic.ca/UVicRevTut/F4rev.html, Department of Computer Science, University of Victoria, 2000.

[Nok99] Nokia Research Center, *Setup and User's Guide to Columbus/CAN Version 2.0*, 17.11.1999.

[Rat00] Rational Software Corporation, *Rational Purify for Windows*, http://www.rational.com/products/purify_nt/, 2000.

[Rat98] Rational Software Corporation, *Rational Rose 98: Roundtrip Engineering with Java*, 1998.

[Rat99a] Rational Software Corporation, *Roundtrip Engineering with Rational Rose and Visual C++*, 1999.

[Rat99b] Rational Software Corporation, *The Unified Modeling Language Notation Guide v.1.3*, http://www.rational.com, January 1999.

[Rat99c] Rational Software Corporation, *OMG Unified Modeling Language Specification Version 1.3*, June 1999.

[Rin00] Rintala Matti, *tutnew library for C++*, http://www.cs.tut.fi/~bitti/tutnew/, 2000.

[RJB99] Rumbaugh J., Jacobson J., and Booch G., *The Unified Modeling Reference Manual*, Addison-Wesley, 1999.

[RN97] Russel Stuart, Norving Peter: *Artificial Intelligence: A Modern Approach*, Prentice Hall, 1994.

[SKS00] Selonen P., Koskimies K., Sakkinen M.: *How to Make Apples from Oranges in UML*. 2000. To appear.

[SS00] Selonen Petri, Systä Tarja: *Synthesizing Annotated Class Diagrams in UML*. 2000. Submitted.

[SSC96] Sefika Mohlalefi, Sane Aamod, Camppbell Roy H., *Architecture-Oriented Visualization*, In *Proceedings Of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA –96)*, ACM Press, 1996, pp. 389-405.

[Sun00] Sun Microsystems, *JavaDoc*, http://java.sun.com/products/jdk/javadoc/, 2000

[Sys00] Systä Tarja, *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*, Department of computer and information sciences University of Tampere 2000

[Wik98] Wikman Johan, Evolution of Distributed Repository-Based Architecture, First Nordic Software Architecture Workshop (NOSA –98), http://www.ide.hk-r.se/~bosch/NOSA98/JohanWikman.pdf, 1998.