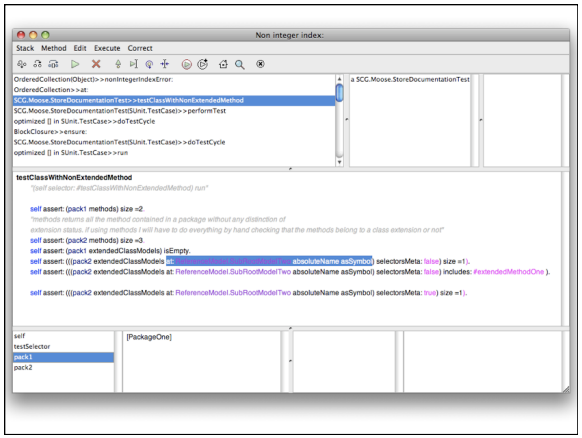
Dynamic Analysis

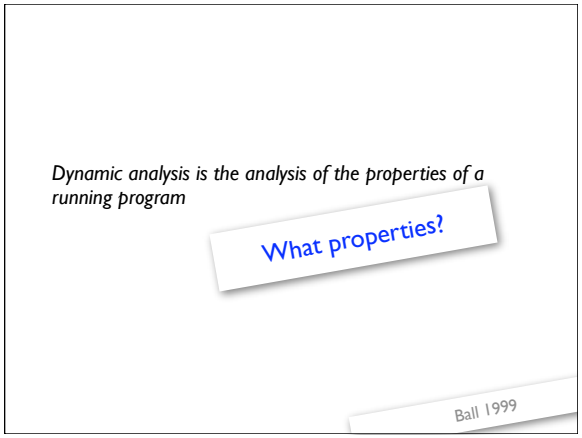
Tudor Gîrba
www.tudorgirba.com

The job of the reverse engineer is similar to the one of the doctor, as they both need to reason about an unknown complex system.

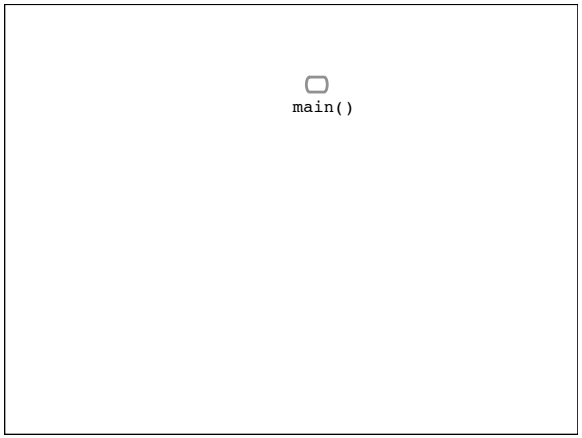
One way to gain insights is to see the system in action. The first thing doctors do in an emergency room is to attach various instruments to the patient to follow the evolution of his condition.



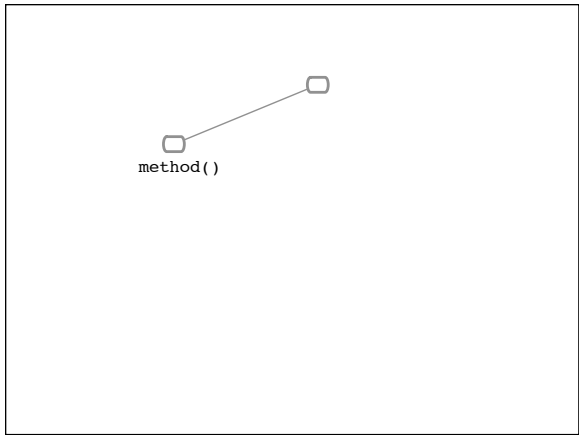
Watching a system run, can reveal a great deal of information about the problem. For example, some reverse engineering patterns mentioned before are related to observing the system run:
- Interview during demo
- Step through the execution
- Tests, your life insurance



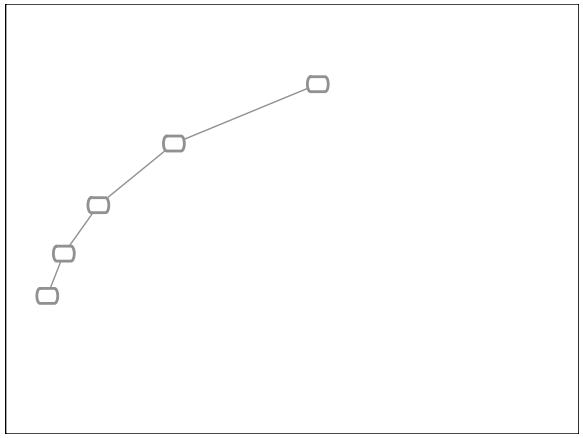
Thomas Ball, “The Concept of Dynamic Analysis,” Proceedings of the European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSC'99), LNCS, no. 1687, Springer Verlag, Heidelberg, September 1999, pp. 216—234.



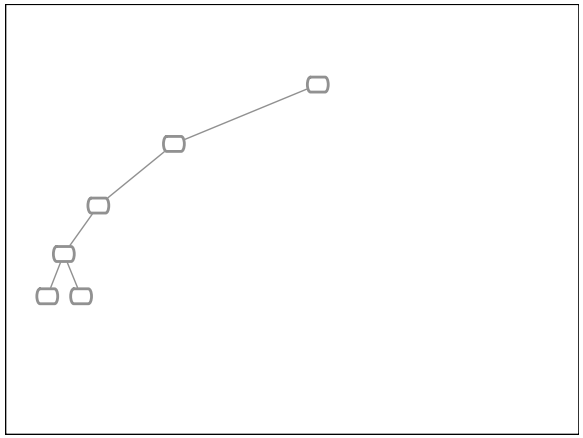
First let’s see what makes the dynamic analysis space. It all starts from the first execution.



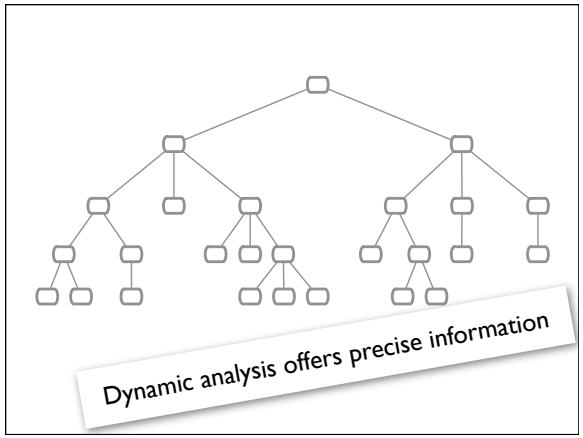
The main method calls a method.



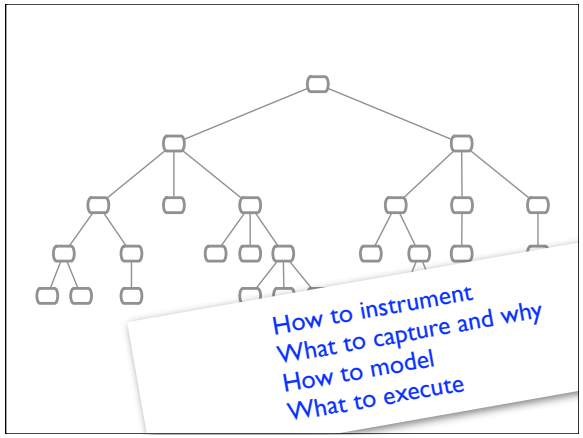
and so on.



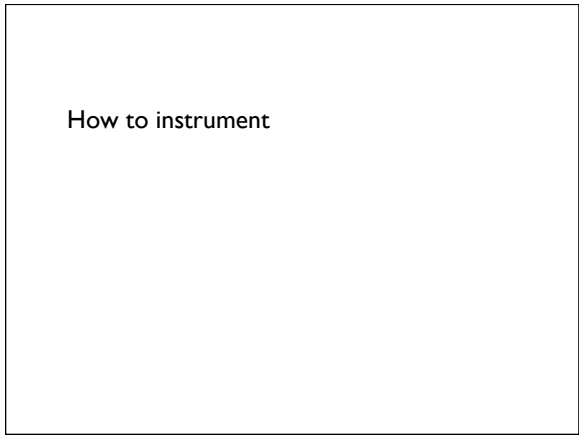
After a method returns, another method is called. Thus, the totality of method executions can be seen as a tree.



Dynamic information offers detailed information about the communication in the system.



Still, the problem is that the dynamic space offers too many details. Thus, we have to know what data to gather and for what purpose. We need to know how to model that data. And we need to know how to obtain the data.



```
…
public class BankAccount {
  private Money balance;

  public void deposit(Money amount) {
     System.out.println("deposit");
     this.balance += money;
  }
}
```

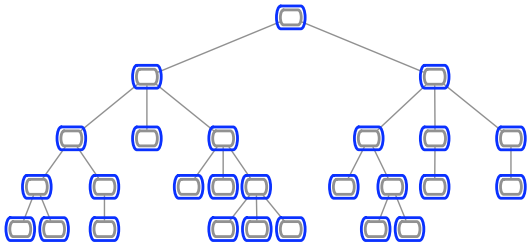
The most primitive way of debugging is to print the state of the system at a certain moment. Let's call this the the poor man's debugging.

```
import org.apache.log4j.Logger;
…
public class BankAccount {
  private Money balance;

  public void deposit(Money amount) {
     logger.info("deposit");
     this.balance += money;
  }
}
```

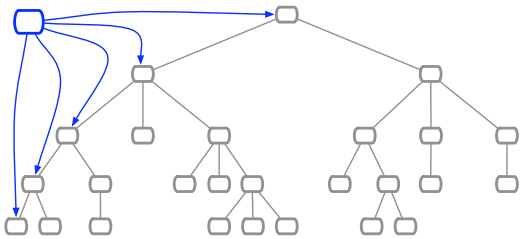
Using a logging framework improves the situation by being able to customize what happens with the data.
Still, we have to add the hooks by hand directly in the code. We also do not keep information about the control flow.

Method Wrappers and Aspects intervene before and after each interesting method



Another possibility is to insert the instrumentation code around the methods of interests. Two solutions are Aspects and Method Wrappers. With Aspects we can inject the instrumentation in the code. Method Wrappers wrap the original methods and thus can be used to control the instrumentation.

Profilers probe the system



Yet another possibility is to run the original program in a process and in another process of higher priority to run a loop that probes the original process.

3 + 4

```
pushConstant: 3
pushConstant: 4
popIntoTemp: 0            “put argument in temp 0”
popIntoTemp: 1            “put receiver in temp 1”
send: +                   “perform addition”
returnTop
```

And yet another possibility is to insert the instrumentation code directly in the bytecode. This example shows the bytecode of a Smalltalk method returning 3+4.

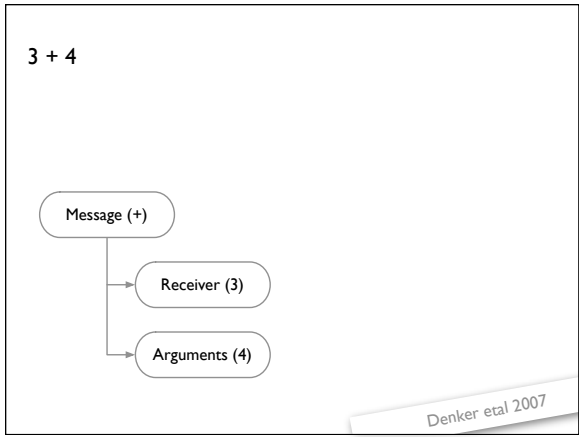
3 + 4
… insertBefore: 'Transcript show: <meta: #receiver>'

```
pushConstant: 3
pushConstant: 4
popIntoTemp: 0            "put argument in temp 0"
popIntoTemp: 1            "put receiver in temp 1"
pushLit: ##Transcript     "start of inserted code"
pushTemp: 1               "push receiver for
                           printing"
send: asString
send: show:
pop                       "end of inserted code"
pushTemp: 1               "rebuild the stack"
pushTemp: 0
send: +
returnTop
```

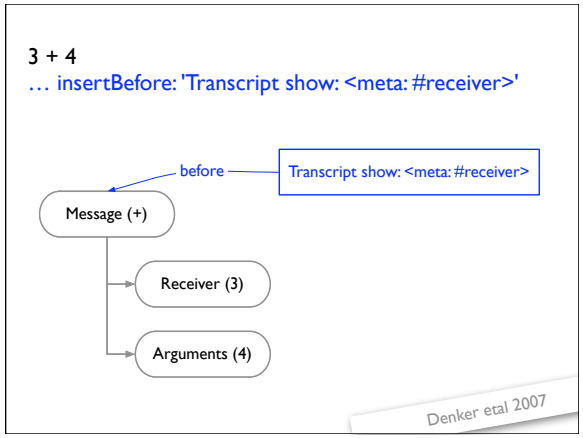
Denker 2008

Marcus Denker, Stéphane Ducasse and Éric Tanter, “Runtime Bytecode Transformation for Smalltalk,” Journal of Computer Languages, Systems and Structures, vol. 32, no. 2-3, July 2006, pp. 125-139.

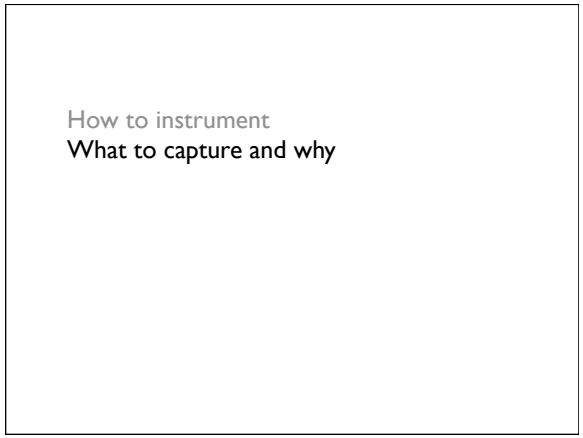
When we want to insert before this method execution a printout of the receiver, the code can look like in the picture. The main benefit here is that we have no intermediary code that needs to be executed (like for example in the case of Method Wrappers). Still, the problem is that

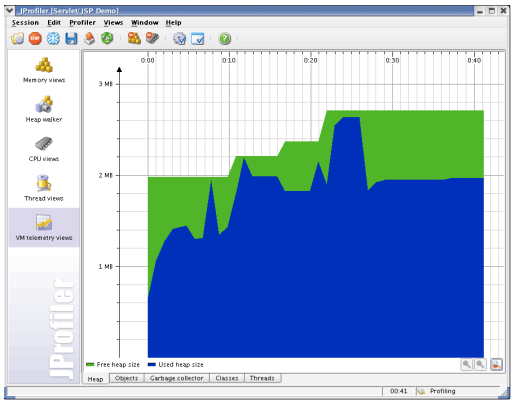


Marcus Denker, Stéphane Ducasse, Adrian Lienhard and Philippe Marschall, “Sub-Method Reflection,” Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007, vol. 6/9, ETH, October 2007, pp. 231—251.



Marcus Denker, Stéphane Ducasse, Adrian Lienhard and Philippe Marschall, “Sub-Method Reflection,” Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007, vol. 6/9, ETH, October 2007, pp. 231—251.



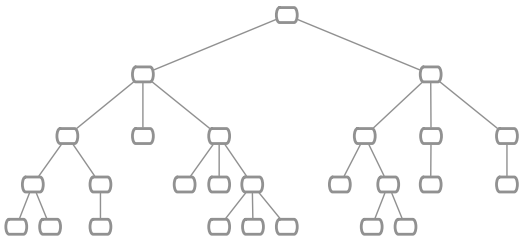


One common use of dynamic analysis is profiling. Targets can be:
- CPU and memory usage (top)
- Network usage (netstat, tcpdump)
- Open files, pipes, sockets (lsof)

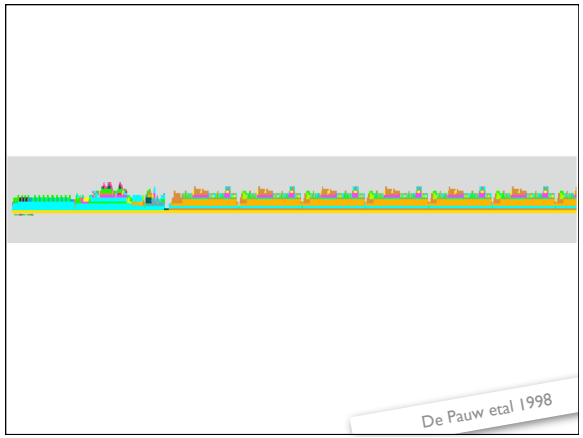
Collecting Garbage is a Dynamic Analysis

Possible implementations are:
- reference counter: it counts the number of references for each object and the objects with 0 references are discarded (rarely used now)
- traverse, mark and clean: another solution is to traverse the memory and mark each objects as they are reached. Those that are not reached get cleaned.

The Control Flow is the most common focus

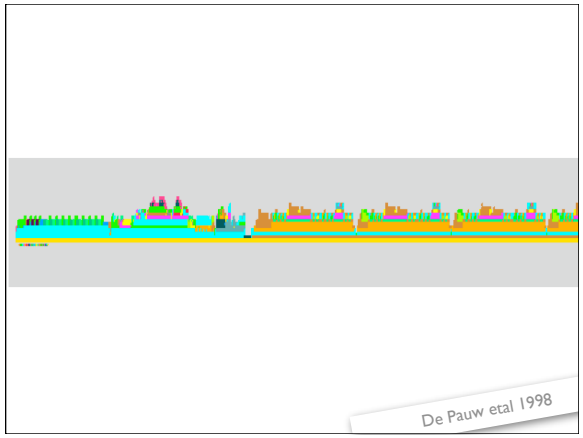


Dynamic analysis comes from procedural programming, that is why many dynamic analyses focus on the control flow.

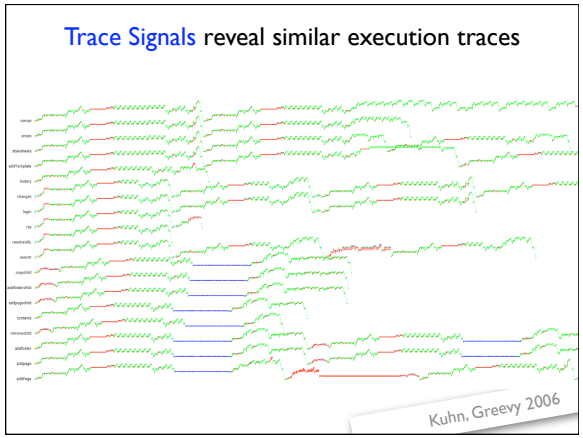


Wim De Pauw, David Lorenz, John Vlissides and Mark Wegman, “Execution Patterns in Object-Oriented Visualization,” Proceedings of Conference on Object-Oriented Technologies and Systems (COOTS'98), USENIX, 1998, pp. 219—234.

The runtime of a system contains tons of data. We need effective means to distill relevant information from this data.

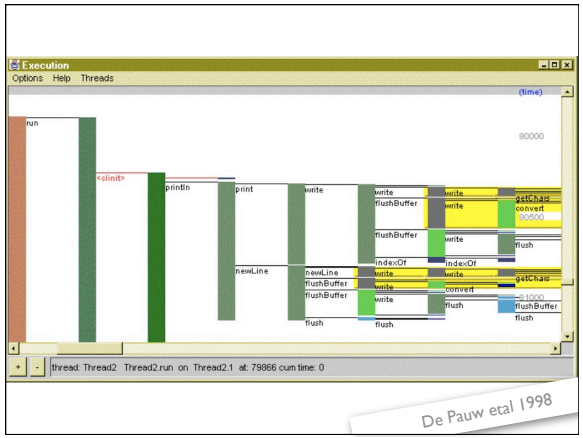


This picture shows a compressed view an execution trace. The orange line at the bottom represents the root of the tree, and the dots from on top represent the leaves of the tree. The colors denote various activation types and reveal repeating patterns.



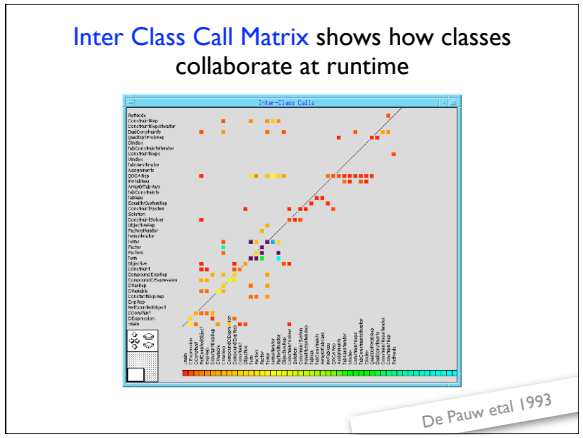
Adrian Kuhn and Orla Greevy, “Exploiting the Analogy Between Traces and Signal Processing,” Proceedings IEEE International Conference on Software Maintainance (ICSM 2006), IEEE Computer Society Press, Los Alamitos CA, September 2006, pp. 320-329.

Another approach summarizes traces into signals (each dot represents an activation) and orders several traces to reveal similarities.



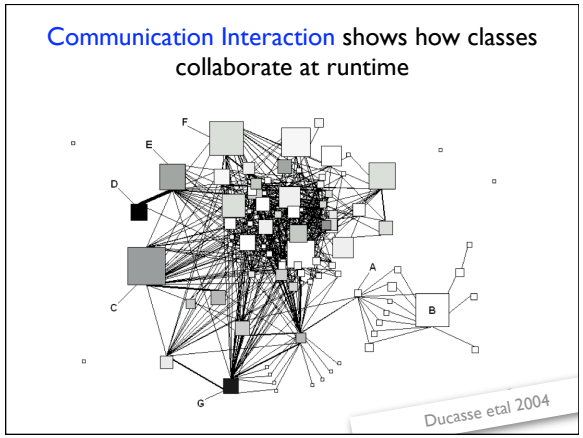
Wim De Pauw, David Lorenz, John Vlissides and Mark Wegman, “Execution Patterns in Object-Oriented Visualization,” Proceedings of Conference on Object-Oriented Technologies and Systems (COOTS'98), USENIX, 1998, pp. 219—234.

Another use case for dynamic analysis is the recovery of sequence diagrams. In this case, activations are related to instances. Such a visualization is only useful when fine grained details are needed.



Wim De Pauw, Richard Helm, Doug Kimelman and John Vlissides, “Visualizing the Behavior of Object-Oriented Systems,” Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93), October 1993, pp. 326—337.

A more coarse grained view is to relate instances to their classes and to recover the runtime communication between classes.

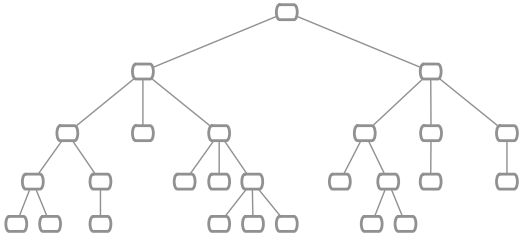


Stéphane Ducasse, Michele Lanza and Roland Bertuli, “High-Level Polymetric Views of Condensed Run-Time Information,” Proceedings of 8th European Conference on Software Maintenance and Reengineering (CSMR'04), IEEE Computer Society Press, Los Alamitos CA, 2004, pp. 309—318.

Another example of class runtime communication. Classes are shown as nodes, and method executions as edges. The graph is arranged according to a spring layout.

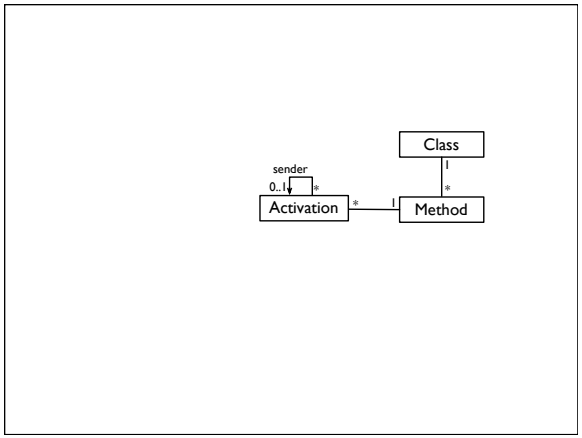
How to instrument
What to capture and why
How to model

A meta-model guides the way we think of a problem domain. What is a good meta-model for dynamic analysis?



This is the space that needs to be modeled.

Each node in the trace is called an Activation.

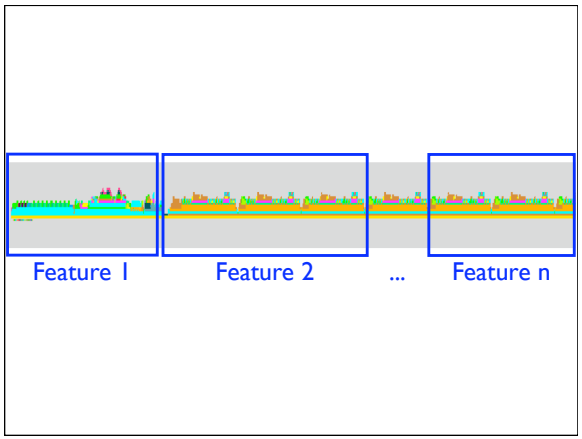


It represents an Activation of a Method. The relationship between Activation and Method allows us to link dynamic information with static information.

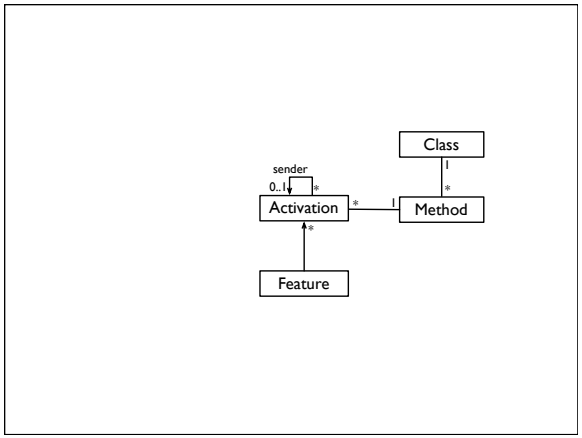
A feature is an observable unit of behavior of a system triggered by the user

Eisenbarth etal 2003

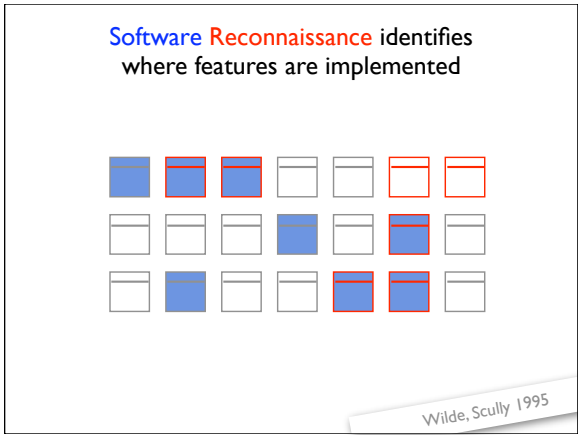
While engineers see internal structure, users see features. One particular opportunity offered by dynamic analysis is to bridge the two worlds by recovering the mapping between features and the code parts that implement them.



We can manually pick the start and end of a feature while we instrument the system.

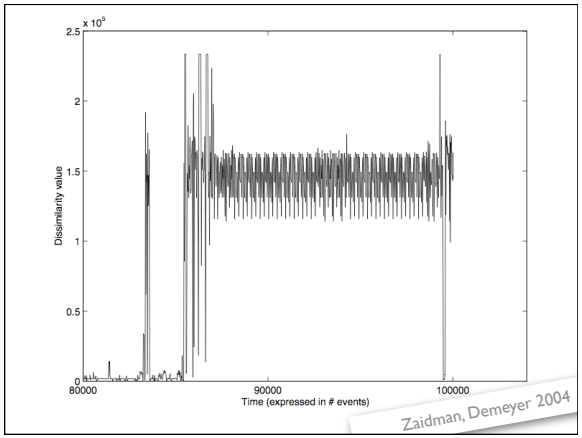


Thus, a Feature is a set of activations.



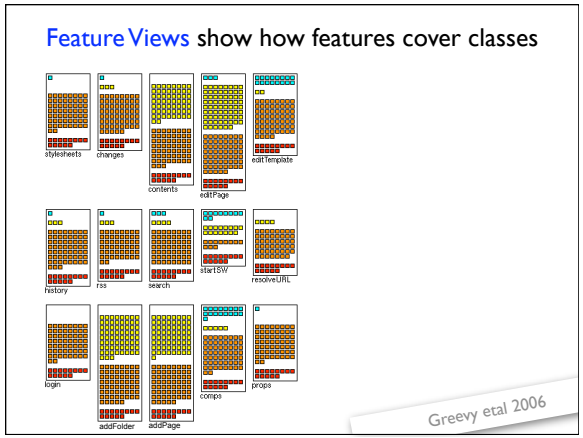
Norman Wilde and Michael Scully, “Software Reconnaisance: Mapping Program Features to Code,” Journal on Software Maintenance: Research and Practice, vol. 7, no. 1, 1995, pp. 49—62.

Software Reconnaisance is a technique to recover the mapping between code parts and features. First, the program is ran by exercising the feature (blue classes), and then it is ran without exercising the feature (red classes). Thus, candidates for most specific classes for a feature are those that appear in the first run, but do not appear in the second one.



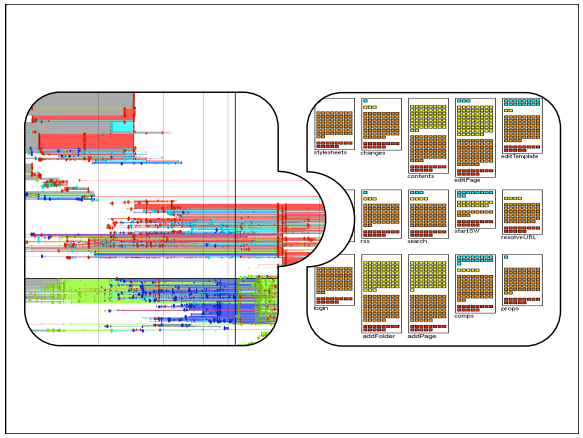
Andy Zaidman and Serge Demeyer, “Managing trace data volume through a heuristical clustering process based on event execution frequency,” Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR'04), IEEE Computer Society Press, Los Alamitos CA, March 2004, pp. 329—338.

In this approach, to identify where to start implementing a new feature is to first identify the most specific part of a similar feature. Shown in the graph is a signal of an execution trace. The signal shows repeating patterns in the execution of the similar feature that can be good starting points.

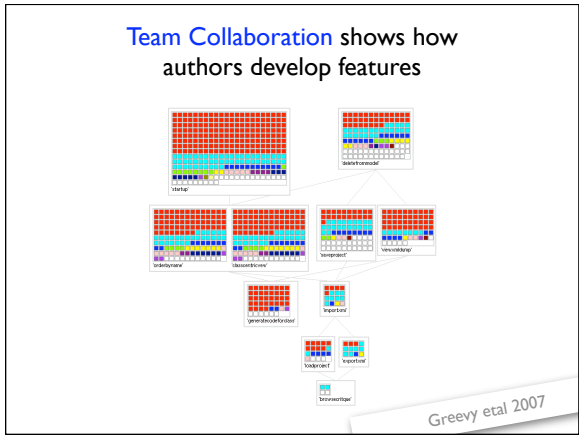


Orla Greevy, Stéphane Ducasse and Tudor Gîrba, “Analyzing Software Evolution through Feature Views,” Journal of Software Maintenance and Evolution: Research and Practice (JSME), vol. 18, no. 6, 2006, pp. 425—456.

In this case, boxes are features and squares are classes. The color of a class is given by the degree of class participation in features:
- cyan denotes classes that only appear in this feature
- yellow denotes classes that appear in less than 50% of features
- orange denotes classes that appear in more than 50% of features
- red denotes classes that appear in all features



Here we show an example of how we can combine two analyses (namely, ownership map and feature views) to obtain identify how developers work on features.



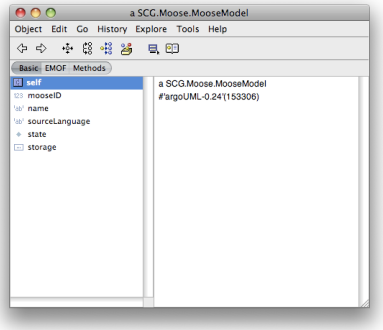
Orla Greevy, Tudor Gîrba and Stéphane Ducasse, “How Developers Develop Features,” Proceedings of 11th European Conference on Software Maintenance and Reengineering (CSMR 2007), IEEE Computer Society, Los Alamitos CA, 2007, pp. 256—274.

In this view, each box represents a feature, and each square represents a class. This time, the color of a class is given by the owner of that class. Thus, the totality of colors in a box represents the team that develops it. The boxes are arranged according to a partial order (the features on top have a large team than those on the bottom).

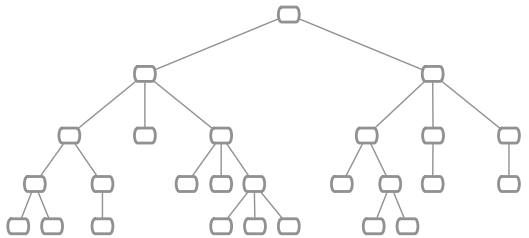
How to instrument
What to capture and why
How to model

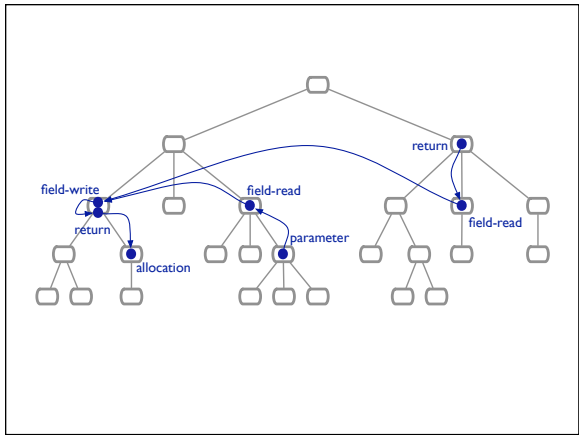
The runtime is more than method activations

Method execution traces do not reveal how
… objects refer to each other
… object references evolve

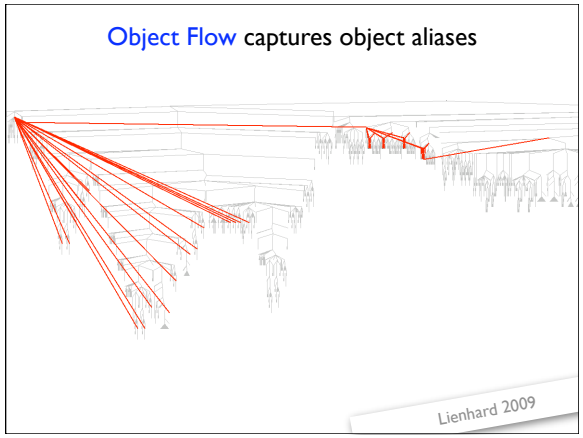


The Smalltalk inspector allows us to check the state of objects at runtime.



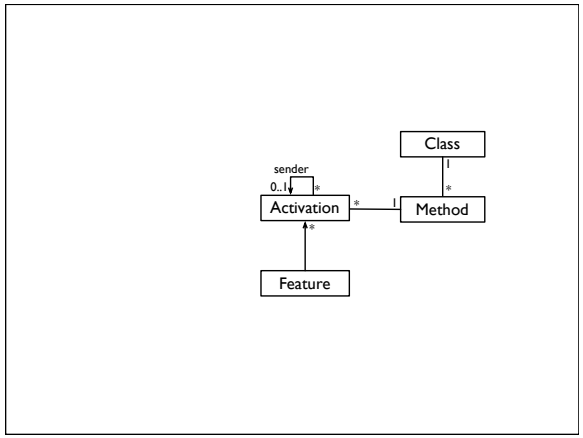


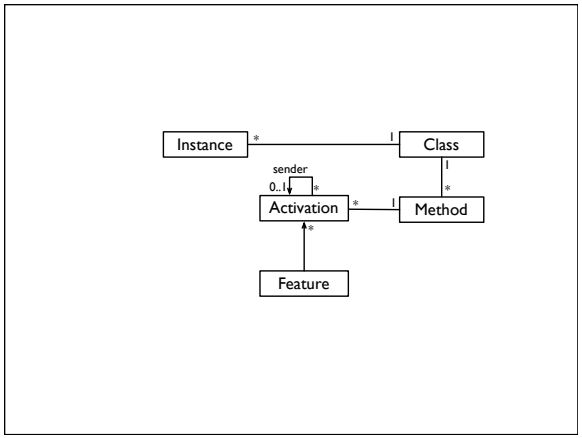
Because of side effects (storing values in instance variables), the way an object gets into a specific state is not obvious.



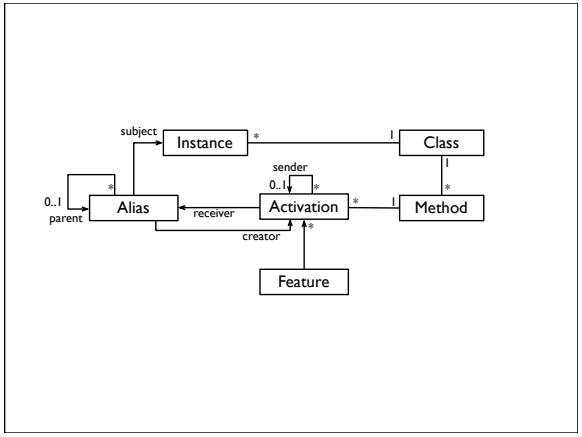
Adrian Lienhard and Stéphane Ducasse and Tudor Gîrba. Taking an Object-Centric View on Dynamic Information with Object Flow Analysis. In Journal of Computer Languages, Systems and Structures 35(1) p. 63--79, 2009.

In this example, the gray tree represents the control flow, while the red tree represents the flow of one object. The object flow intertwines the control flow and tells a complementary story.



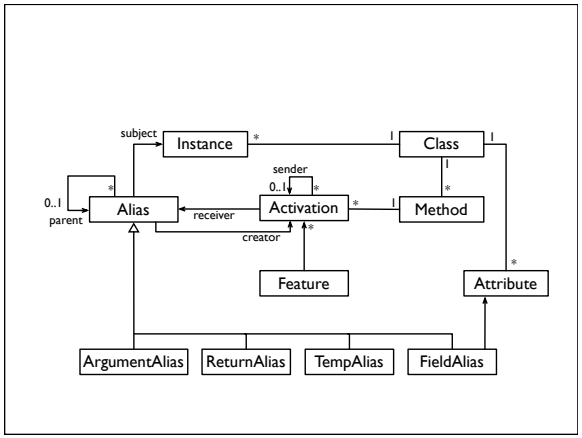


To model how objects flow, we first model instances.



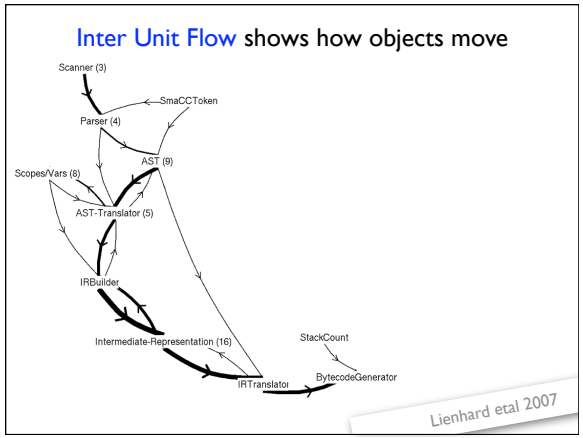
Aliases model a reference to an Instance. An alias is obtained from a possible parent Alias. Thus, Aliases form a tree.

Both the receiver of an Activation and its creator are not represented directly by Instances but by Aliases. As such, we can know exactly where an alias was used.



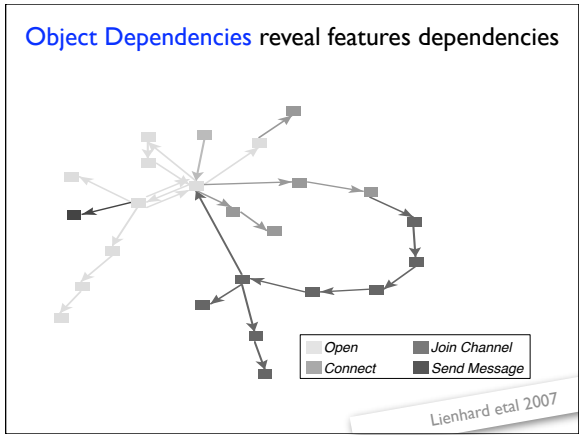
There can be several types of creating an alias:
- by passing it via an argument
- by returning it from a method
- by storing in/reading from a temp variable
- by storing in/reading from a field

The field alias is shown as linked to its static counter part, Attribute. Similar relationships exist for ArgumentAliases and TemoAliases.



Adrian Lienhard, Stéphane Ducasse and Tudor Gîrba, “Object Flow Analysis — Taking an Object-Centric View on Dynamic Analysis,” Proceedings of the 2007 International Conference on Dynamic Languages (ICDL'07), ACM Digital Library, New York, NY, USA, 2007, pp. 121—140.

The view shows modules or units as nodes and the amount of objects transmitted as edges. The system under study is an compiler: the objects flow from the Scanner to the Parser, then to the Intermediate Representation Builder and finally to the ByteCodeGenerator.

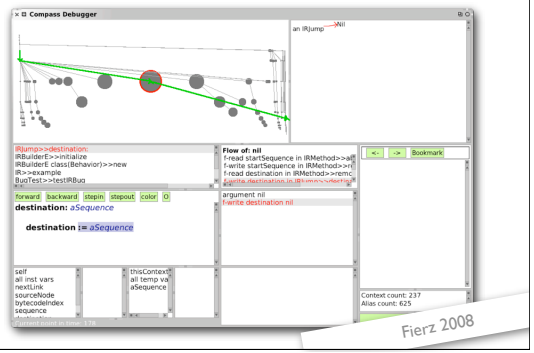


drian Lienhard, Orla Greevy and Oscar Nierstrasz, “Tracking Objects to detect Feature Dependencies,” Proceedings of International Conference on Program Comprehension (ICPC'07), IEEE Computer Society, Washington, DC, USA, June 2007, pp. 59—68.

This view shows objects and references. The color denotes the moment of the creation of the object. The scale of gray is given by the different features exercised. In this example, the system is an IRC client and the features are executed in order: open, connect, join channel, send message to the channel.

Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken and Michael I. Jordan, “Scalable statistical bug isolation,” Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05), ACM, New York, NY, USA, 2005, pp. 15—26.

Back in time debuggers remember more than the current stack



The screenshot is taken from the Compass prototype developed by Julien Fierz.

How to instrument
What to capture and why
How to model
What to execute

The runtime is more than method activations

To interpret the result of a dynamic analysis, you have to control what is executed.

Tudor Gîrba
www.tudorgirba.com