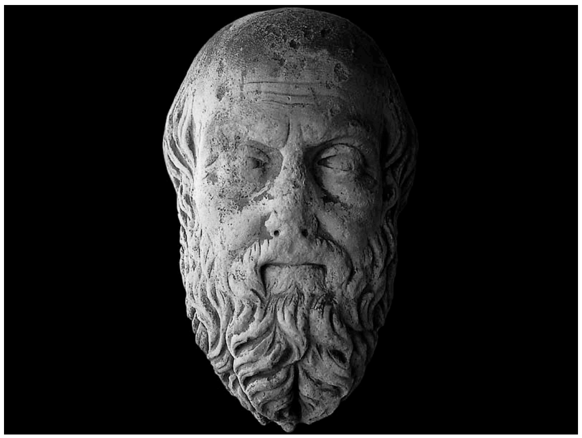
History analysis

Tudor Gîrba
www.tudorgirba.com

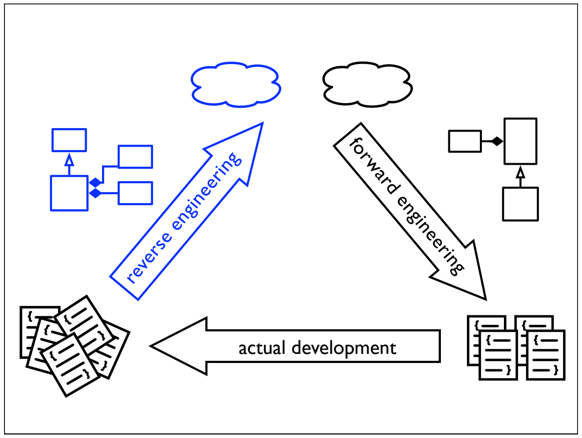
UNIVERSITÄT BERN



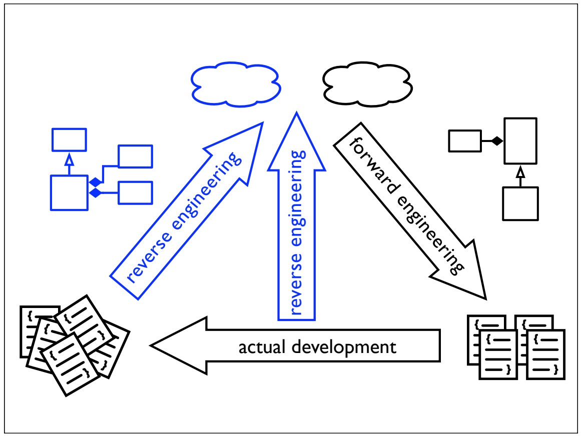
http://en.wikipedia.org/wiki/Herodotus
History comes from the Greek word ἱστορίαι meaning inquiry.

Herodotus … displays his enquiry, so that human achievements may not become forgotten … and great and marvelous deeds … may not be without their glory … especially to show why the two peoples fought with each other.

Herodotus



Reverse engineering is needed to re-synchronize the original idea with the reality of the implementation.

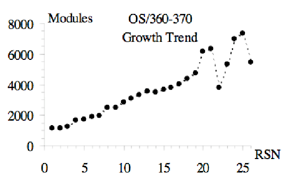


The way the system got in the current shape is also relevant for understanding the current situation.

History holds useful information

History holds useful information

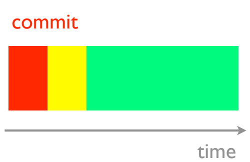
When did it change?

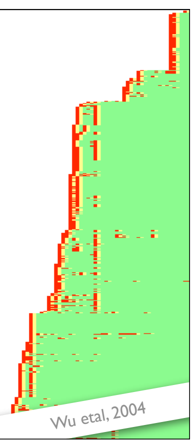


Lehman etal, 2001

A classic approach to take a look at evolution is to use a line chart showing how one variable changed over time. The graph presented here contributed to the formulation of the Lehman's Laws of Evolution.

Spectographs show change activity





Wu etal, 2004

Jingwei Wu, Richard Holt and Ahmed Hassan, "Exploring Software Evolution Using Spectrographs," Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004), IEEE Computer Society Press, Los Alamitos CA, November 2004, pp. 80-89.

Spectographs show the history of files or modules, ordered by life span: the newest is on top.

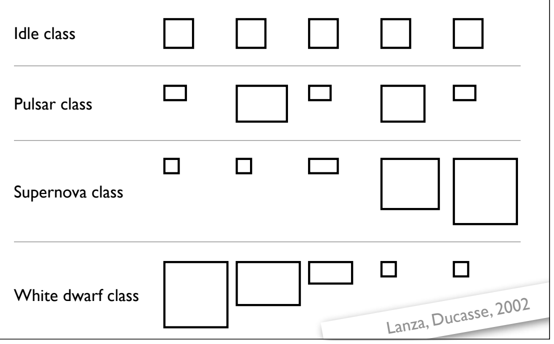
Red marks the change activity (i.e., commits in CVS). Red transforms into yellow to show the age of activity, and eventually turns into green to show no recent activity.

History holds useful information

When did it change?

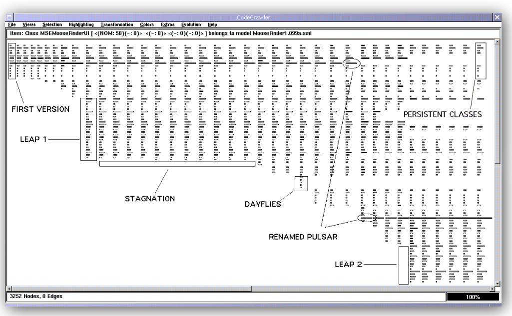
How did it change?

Evolution Matrix shows changes in classes



Michele Lanza and Stéphane Ducasse, “Understanding Software Evolution Using a Combination of Software Visualization and Software Metrics,” Proceedings of Langages et Modèles à Objets (LMO 2002), Lavoisier, Paris, 2002, pp. 135-149. http://www.iam.unibe.ch/~scg/cgi-bin/scgbib.cgi/abstract=yes?Lanz02a

The Evolution Matrix shows classes as boxes (width = NOA, height = NOM). By arranging the classes in rows, patterns of evolution can be spotted.



Taking a global view on the system, we can also spot when classes were introduced and how long they lived.

History holds useful information

When did it change?
How did it change?
What changed?

2	4	3	5	7
2	2	3	4	9
2	2	1	2	3
2	2	2	2	2
1	5	3	4	4

Suppose we have 5 classes and their respective number of methods throughout 5 versions. Which one changed the most?

Evolution of
Number of Methods

$$ENOM(C) = \sum |NOM_i(C)-NOM_{i-1}(C)|$$

$$ENOM(C) = 4 + 2 + 1 + 0 = 7$$

1	5	3	4	4

Evolution of a Property counts the sum of the absolute changes of a Property in subsequent versions.

Latest Evolution of Number of Methods

$$LENOM(C) = \sum |NOM_i(C)-NOM_{i-1}(C)|\, 2^{i-n}$$

Earliest Evolution of Number of Methods

$$EENOM(C) = \sum |NOM_i(C)-NOM_{i-1}(C)|\, 2^{2-i}$$

$$LENOM(C) = 4\, 2^{-3} + 2\, 2^{-2} + 1\, 2^{-1} + 0\, 2^{0} = 1.5$$

1	5	3	4	4

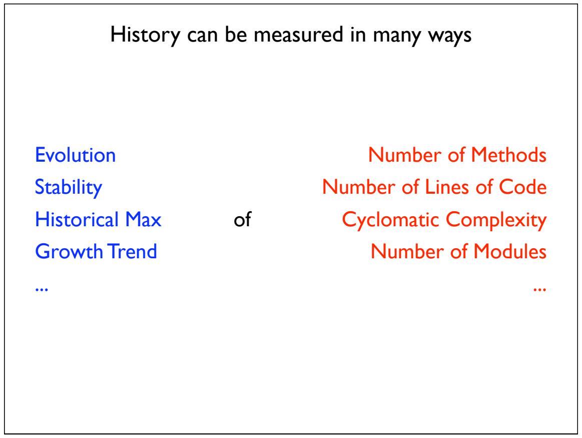
$$EENOM(C) = 4\, 2^{0} + 2\, 2^{-1} + 1\, 2^{-2} + 0\, 2^{-3} = 5.25$$

Latest and Earliest Evolution of a Property put emphasis on the latest or earliest period.

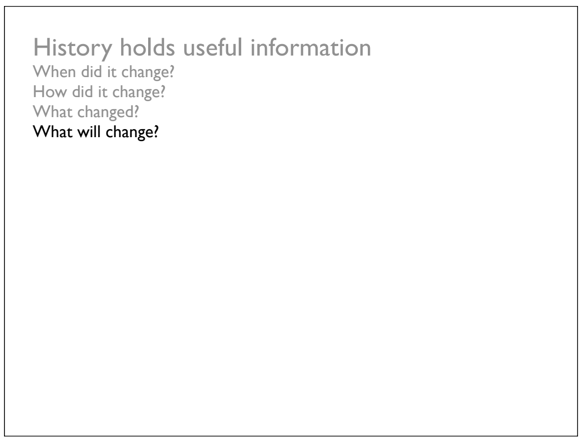
	ENOM	LENOM	EENOM
2 4 3 5 7	7	3.5	3.25
2 2 3 4 9	7	5.75	1.37
2 2 1 2 3	3	1	2
2 2 2 2 2	0	0	0
1 5 3 4 4	7	1.25	5.25

	ENOM	LENOM	EENOM
balanced changer	7	3.5	3.25
late changer	7	5.75	1.37
	3	1	2
dead stable	0	0	0
early changer	7	1.25	5.25

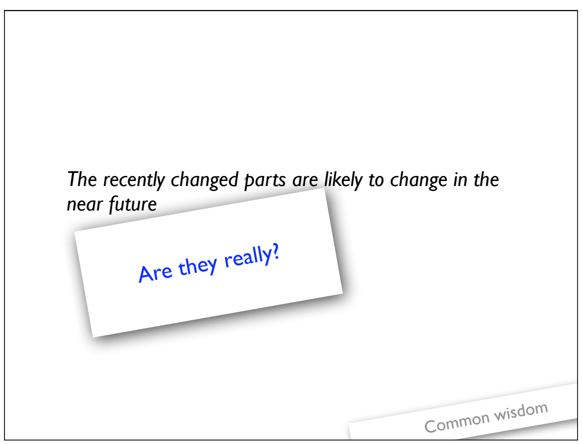
Historical measurements summarize the evolution details.



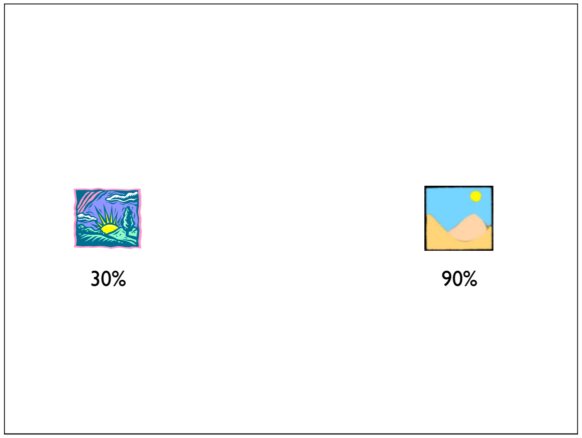
History can be measured in many ways. Still, a metric is just a tool that should be used to answer a question. Starting from the question makes it clearer on what metrics make sense.



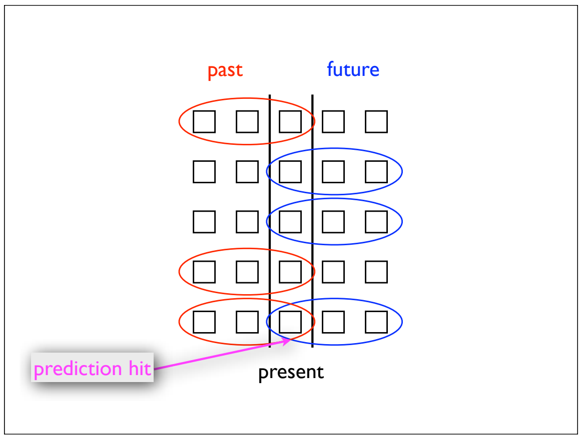
“Those who cannot learn from history are doomed to repeat it.” said George Santayana. In our case, we study history exactly to see what will get repeated and what not.



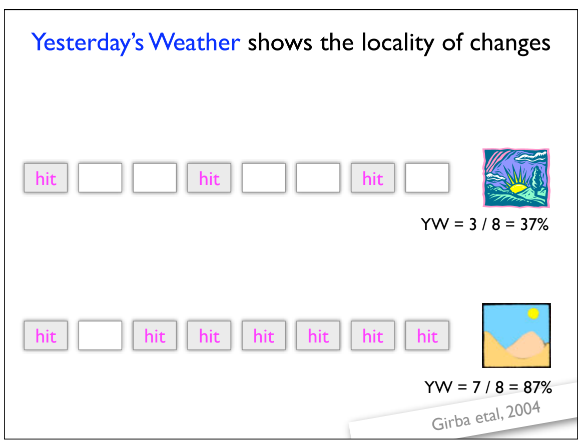
One common wisdom says to start the reverse engineering efforts from the parts that were changed the most lately. But, is it really the case in all systems?



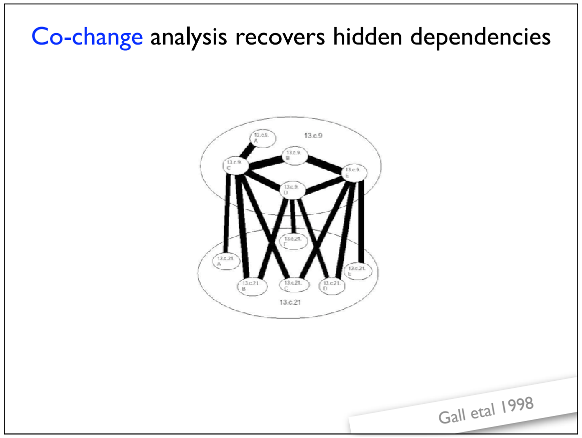
In many places, a good heuristic for predicting today's weather is to say that it is similar to yesterday's weather. However, in other places, the weather changes more often, and the heuristic would fail. Thus, this heuristic is place specific.



Having the history of the software system at hand, we can choose any version to be the present one and thus to check the validity of the Yesterday's Weather heuristic on the current system. If at least one of the entities that changed the most lately is among those that will change the most in the near future, the heuristic produces a hit for that respective version.

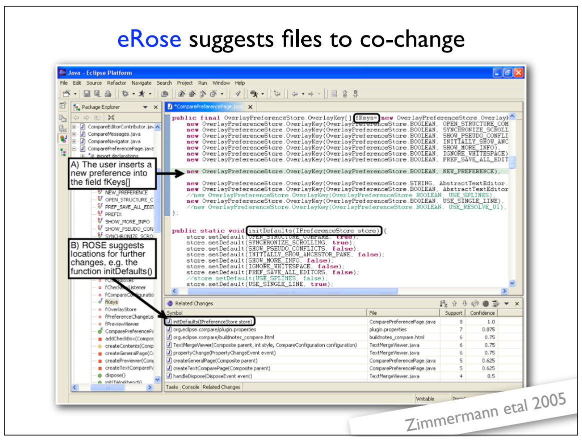


We can then apply on all versions and compute an average to identity the relevance of the heuristic. If it is high enough, we should use it on the system. Otherwise it is not relevant to use it.



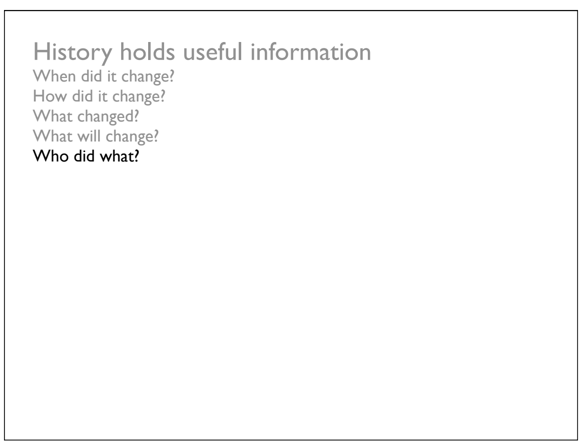
Harald Gall, Karin Hajek and Mehdi Jazayeri, “Detection of Logical Coupling Based on Product Release History,” Proceedings International Conference on Software Maintenance (ICSM '98), IEEE Computer Society Press, Los Alamitos CA, 1998, pp. 190—198.

Co-changes are relationships that can be observed only in time, as they appear when two entities are committed repeatedly in the same time.

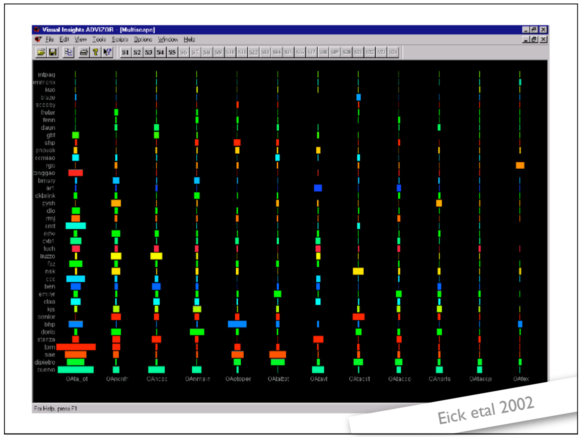


Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, Andreas Zeller. Mining Version Histories to Guide Software Changes. In IEEE Transactions on Software Engineering(31): 429-445 (2005), June 2005, pp. 429-445.

eRose is a tool that reveals files that have been co-changed with the current file, thus offering recommendations related to what else should be changed in the system.

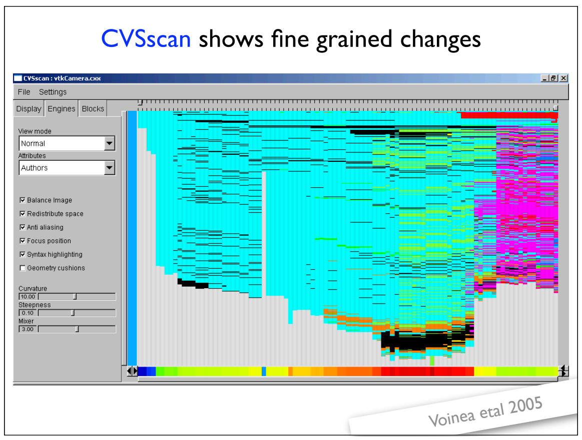


Software is developed by people. History holds information of who did what. To get answers, we need to know who to ask a certain question.



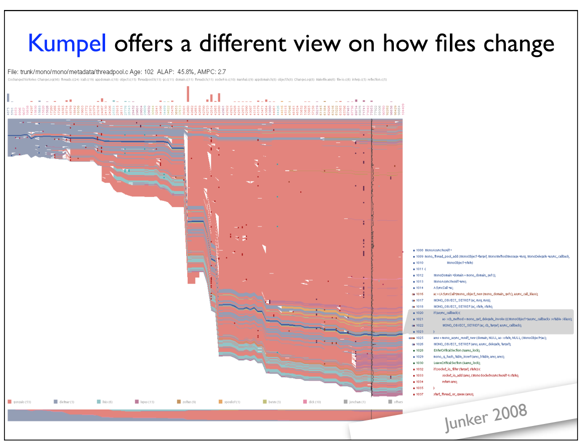
Stephen Eick, Todd Graves, Alan Karr, Audris Mockus and Paul Schuster, “Visualizing Software Changes,” IEEE Transactions on Software Engineering, vol. 28, no. 4, 2002, pp. 396—412.

This visualization displays the correlation of authors (on the rows) and modules (on the columns). Each cell shows the impact of an author on the module.



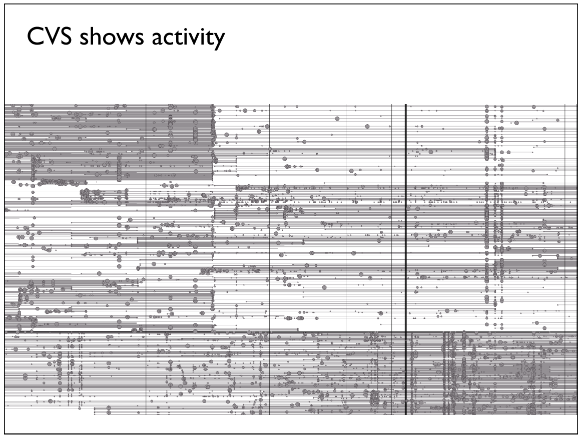
Lucian Voinea, Alex Telea and Jarke J. van Wijk, “CVSscan: visualization of code evolution,” Proceedings of 2005 ACM Symposium on Software Visualization (Softviz 2005), St. Louis, Missouri, USA, May 2005, pp. 47—56.

CVSscan shows fine grained information about how a file evolved.



http://moose.unibe.ch/tools/yellowsubmarine

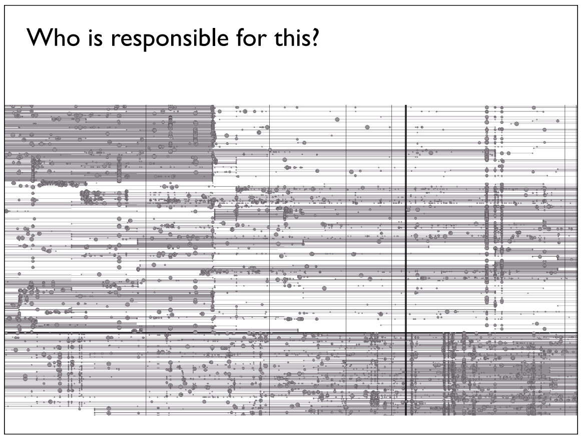
Kumpel is an interactive visualization for browsing the history of files.



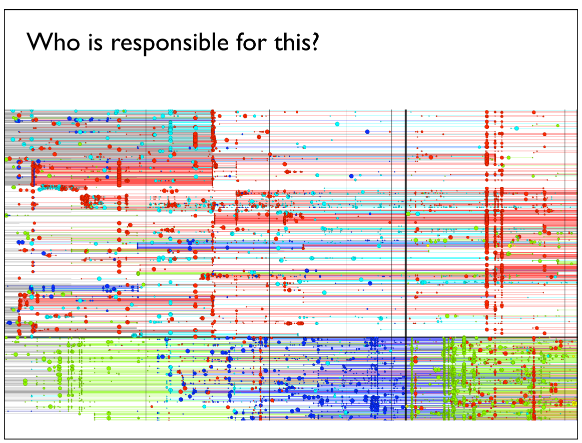
Let’s take a look at the process of creating two visualizations. The first one is to learn from CVS who worked where, when and with whom.

The picture shows files as lines, and commits as circles on the lines.

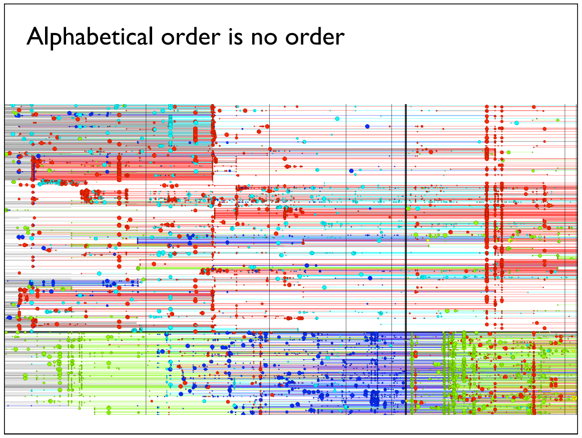
The files are split into two parts: the upper part shows the Java files, and the lower part shows the JSP files.
Inside each part the files are ordered alphabetically.



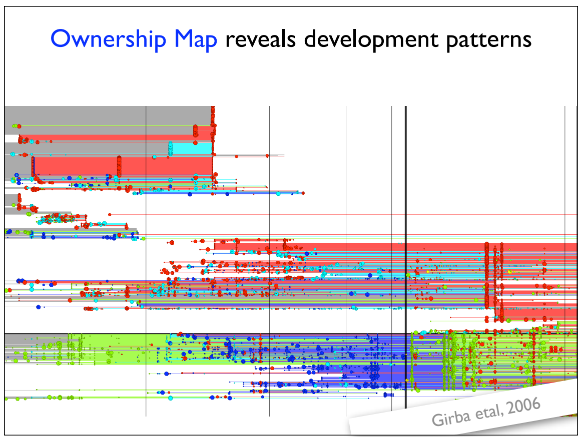
But, who did what and when?



We color each file by the developer that wrote the most lines of code in a certain period.

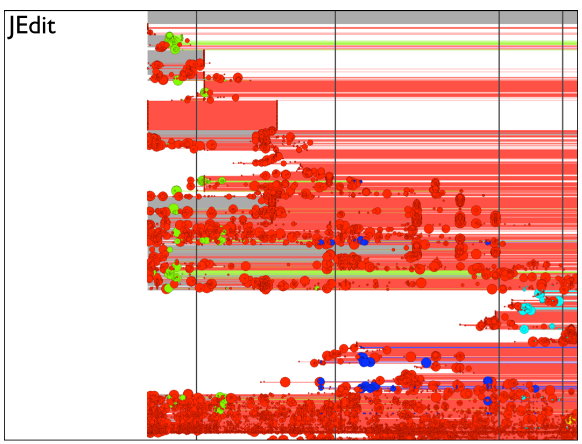


The files are ordered alphabetically, but in this case pure alphabetical order is not a significant order when patterns of activity are the target. Even so, it can be seen that red is mainly in charge with Java (upper part), and blue and green with JSP (lower part).

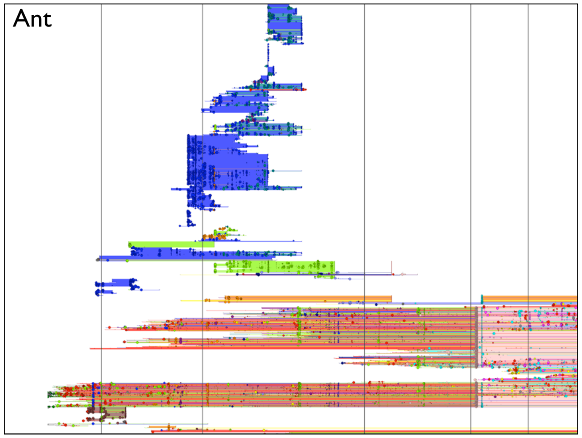


Tudor Gîrba, Adrian Kuhn, Mauricio Seeberger and Stéphane Ducasse, "How Developers Drive Software Evolution," Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2005), IEEE Computer Society Press, 2005, pp. 113—122. http://www.iam.unibe.ch/~scg/cgi-bin/scgbib.cgi/abstract=yes?Girb05c

The lines are ordered according to their commit signatures: those that have similar commit patterns are placed near each other.
The picture reveals:
- the JSP part was mainly developed by green in the beginning. Afterwards, green left and blue entered the project and started to familiarize himself with the project and then extended it.
- green eventually came back in the project and took over from blue



JEdit is mainly developed by one author.



The blue part was developed mainly by one author and at some point it was removed from Ant and became a project of its own.

Who copied from whom?

```
(john    23.06.03) public boolean stillValid (ToDoItem I, Designer dsgr) {
(bill    09.01.05)    if (!isActive()) {
(bill    09.01.05)       return false
(bill    09.01.05)    }
(steve   16.02.05)    List offs = i.getOffenders();
(john    23.06.03)    Object dm = offs.firstElement();
(steve   16.02.05)    ListSet newOffs = computeOffenders(dm);
(john    23.06.03)    boolean res = offs.equals(newOffs);
(john    23.06.03)    return res;


(george  13.02.05) public boolean stillValid (ToDoItem I, Designer dsgr) {
(bill    11.13.05)    if (!isActive()) {
(bill    11.13.05)       return false
(bill    11.13.05)    }
(steve   16.02.05)    List offs = i.getOffenders();
(george  13.02.05)    Object dm = offs.firstElement();
(steve   16.02.05)    ListSet newOffs = computeOffenders(dm);
(george  13.02.05)    boolean res = offs.equals(newOffs);
(george  13.02.05)    return res;
```

The "cvs annotate" command annotates each line of a file with author that perform the last change and date of the change.
We can use this information to identify who copies from whom.

Who copied from whom?

```
(john    23.06.03) public boolean stillValid (ToDoItem I, Designer dsgr) {
(bill    09.01.05)    if (!isActive()) {
(bill    09.01.05)       return false
(bill    09.01.05)    }
(steve   16.02.05)    List offs = i.getOffenders();
(john    23.06.03)    Object dm = offs.firstElement();
(steve   16.02.05)    ListSet newOffs = computeOffenders(dm);
(john    23.06.03)    boolean res = offs.equals(newOffs);
(john    23.06.03)    return res;


(george  13.02.05) public boolean stillValid (ToDoItem I, Designer dsgr) {
(bill    11.13.05)    if (!isActive()) {
(bill    11.13.05)       return false
(bill    11.13.05)    }
(steve   16.02.05)    List offs = i.getOffenders();
(george  13.02.05)    Object dm = offs.firstElement();
(steve   16.02.05)    ListSet newOffs = computeOffenders(dm);
(george  13.02.05)    boolean res = offs.equals(newOffs);
(george  13.02.05)    return res;
```

We color the lines by the author.

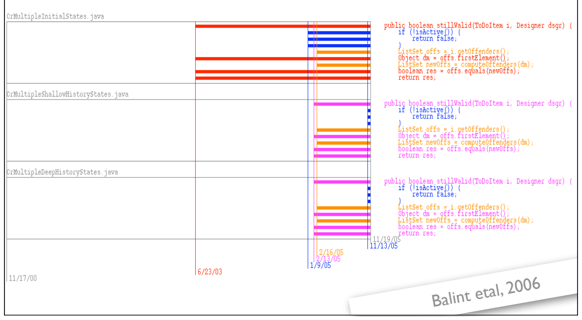
When did changes happen?

```
23.06.03          public boolean stillValid (ToDoItem I, Designer dsgr) {
09.01.05             if (!isActive()) {
09.01.05                return false
09.01.05             }
16.02.05             List offs = i.getOffenders();
23.06.03             Object dm = offs.firstElement();
16.02.05             ListSet newOffs = computeOffenders(dm);
23.06.03             boolean res = offs.equals(newOffs);
23.06.03             return res;


13.02.05          public boolean stillValid (ToDoItem I, Designer dsgr) {
11.13.05             if (!isActive()) {
11.13.05                return false
11.13.05             }
16.02.05             List offs = i.getOffenders();
13.02.05             Object dm = offs.firstElement();
16.02.05             ListSet newOffs = computeOffenders(dm);
13.02.05             boolean res = offs.equals(newOffs);
13.02.05             return res;
```

We remove the author name, because we have the information in the color.

Clone Evolution shows how developers copy



Mihai Balint, Tudor Gîrba and Radu Marinescu, “How Developers Copy,” Proceedings of International Conference on Program Comprehension (ICPC 2006), 2006, pp. 56—65. http://www.iam.unibe.ch/~scg/cgi-bin/scgbib.cgi/abstract=yes?Bali06a

The left hand side shows the changes placed in the overall context of the project time span. To emphasize the order and to show the date of the change, a vertical line is drawn for each date and the date is written below. This is especially useful when we need to distinguish between close changes. The fragments are also ordered so that the original is on top. The name of the containing file is also shown.

The picture reveals that red wrote the original code and blue changed/added 3 lines, pink duplicated the code, orange changed consistently in all three fragments

History holds useful information

When did it change?
How did it change?
What changed?
What will change?
Who did what?

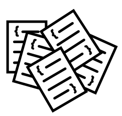
How to model history?

History holds useful information

When did it change?
How did it change?
What changed?
What will change?
Who did what?

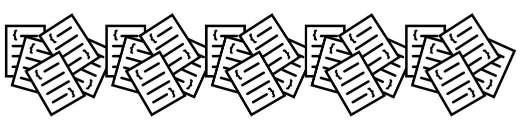
How to model history?

How to model structure changes?



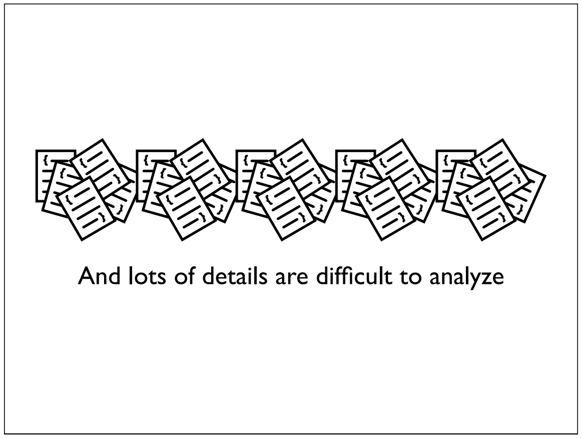
A large system contains lots of details

First, why do we need to know how to model history?

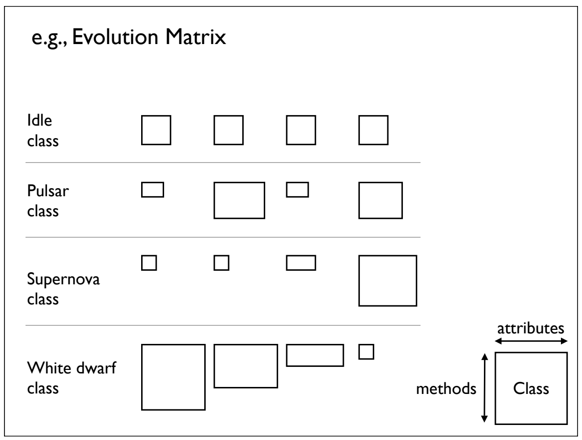


Its history contains even more details

Yes, and?

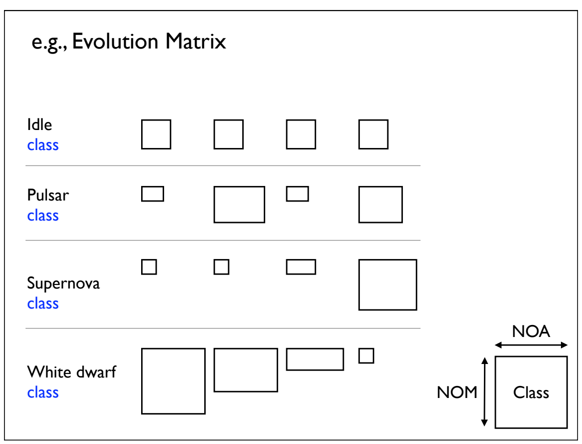


Many details pose two problems. First, they pose a computational problem. Luckily this is solved by Moore's Law. Second, it's an analysis problem because we need to find and interpret the right details for the problem at hand. Thus, we need to know how to tackle these details.

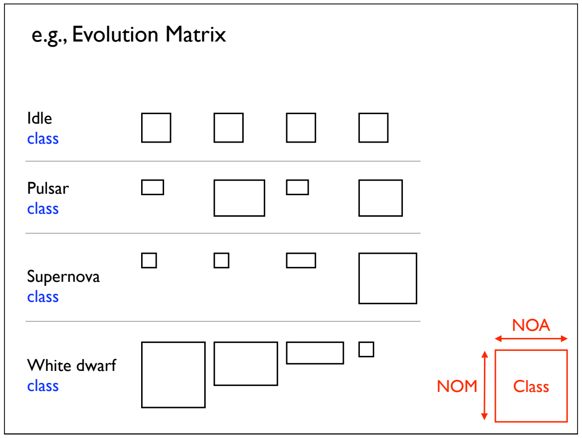


Let's take a closer look at the Evolution Matrix.

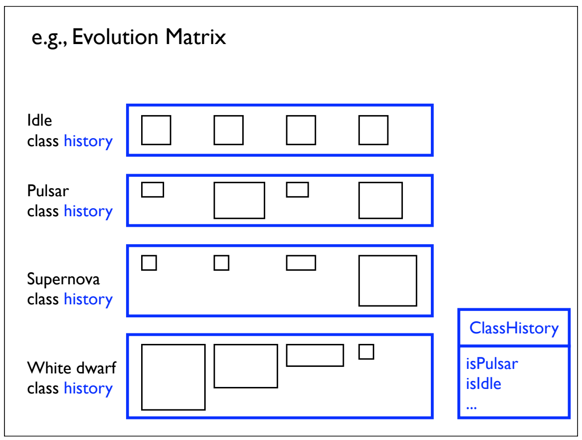
Michele Lanza and Stéphane Ducasse, "Understanding Software Evolution Using a Combination of Software Visualization and Software Metrics," Proceedings of Langages et Modèles à Objets (LMO 2002), Lavoisier, Paris, 2002, pp. 135-149. http://www.iam.unibe.ch/~scg/cgi-bin/scgbib.cgi/abstract=yes?Lanz02a



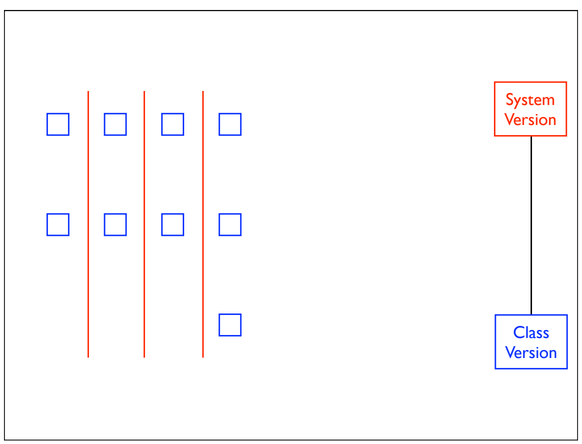
The Evolution Matrix reveals class evolution patterns like supernova or pulsar.



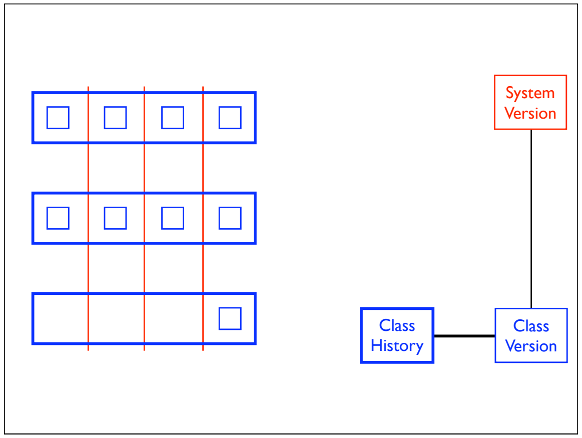
The question is if the “class” from the legend represents the same concept as the one described on the left. The answer is no.



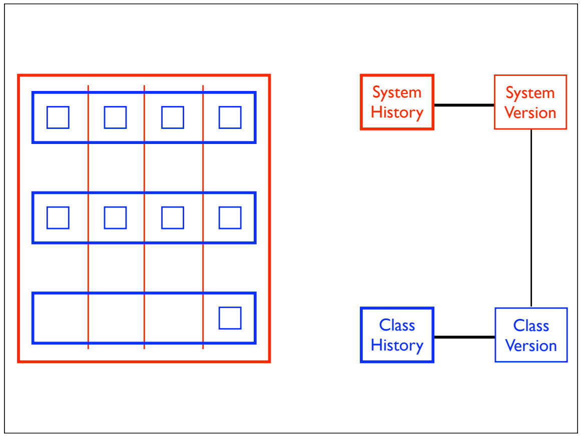
We introduce history as first class entity to encapsulate the evolution of entities (in this case classes).



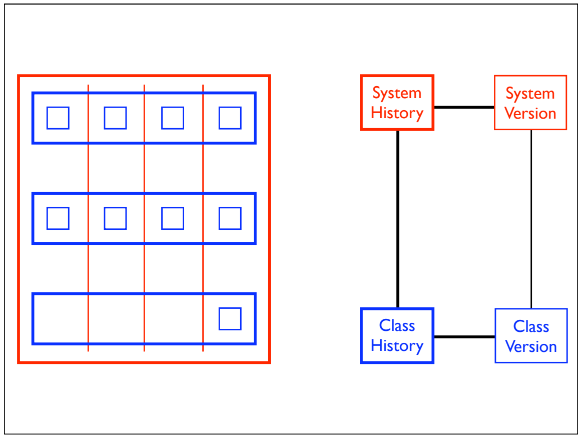
To the left we have actual classes spanning over 4 versions. To the right we have the meta-model. Structurally, we can say that in a SystemVersion there are several ClassVersions.



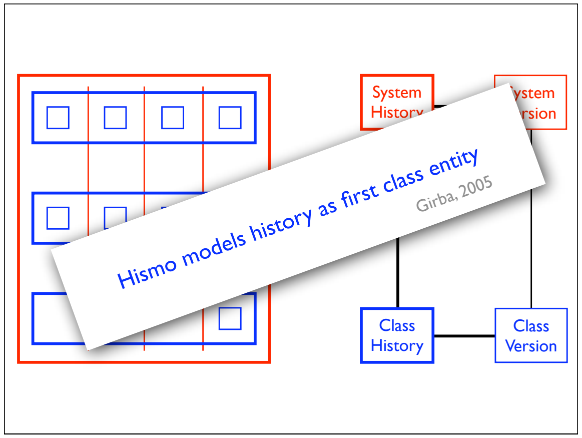
A ClassHistory is formed by several ClassVersions.



The entire picture forms the SystemHistory.



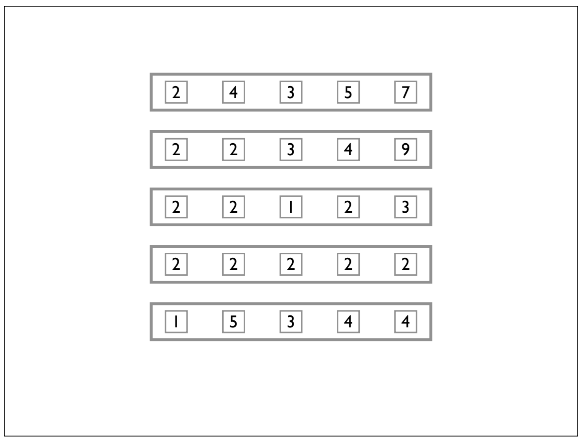
Graphically, inside the large red rectangle (representing the SystemHistory) we have several large blue rectangles (representing ClassHistories). Thus, we can say that in a SystemHistory we have several ClassHistories.



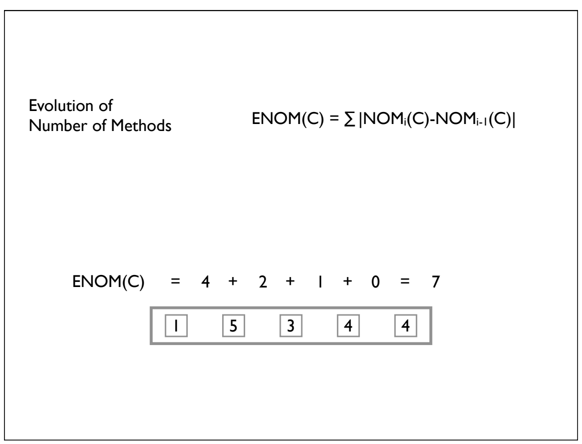
Tudor Gîrba, “Modeling History to Understand Software Evolution,” Ph.D. thesis, University of Bern, Bern, November 2005.

Tudor Gîrba and Stéphane Ducasse, “Modeling History to Analyze Software Evolution,” Journal of Software Maintenance: Research and Practice (JSME), vol. 18, 2006, pp. 207—236.

Hismo stands for History Meta-model.



History can be measured.



Evolution of a Property is a historical measurement.

Latest Evolution of Number of Methods

$LENOM(C) = \sum |NOM_i(C)-NOM_{i-1}(C)|\ 2^{i-n}$

Earliest Evolution of Number of Methods

$EENOM(C) = \sum |NOM_i(C)-NOM_{i-1}(C)|\ 2^{2-i}$

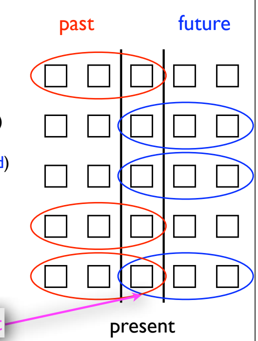
$LENOM(C) = 4\ 2^{-3} + 2\ 2^{-2} + 1\ 2^{-1} + 0\ 2^{0} = 1.5$

1	5	3	4	4

$EENOM(C) = 4\ 2^{0} + 2\ 2^{-1} + 1\ 2^{-2} + 0\ 2^{-3} = 5.25$

The same for Latest Evolution and Earliest Evolution.

```
YesterdayWeatherHit(present):
  past:=histories.topLENOM(start, present)
  future:=histories.topEENOM(present, end)
  past.intersectWith(future).notEmpty()
```



prediction hit

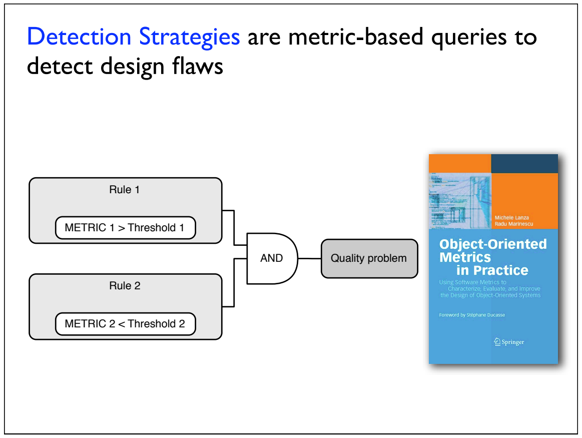
Having these measurements characterizing the history, the code for Yesterday's Weather becomes trivial.

History holds useful information

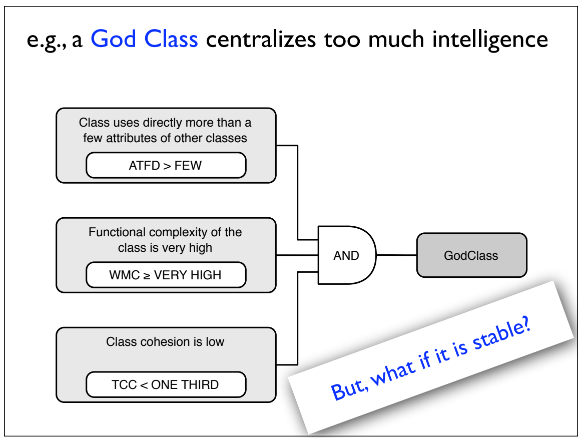
When did it change?
How did it change?
What changed?
What will change?
Who did what?

How to model history?

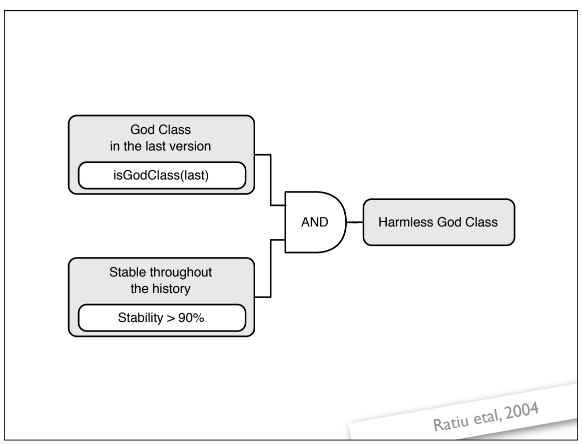
How to model structure changes?
How to combine time with structure?



Michele Lanza and Radu Marinescu, Object-Oriented Metrics in Practice, Springer-Verlag, 2006.



Intuition tells us to eradicate GodClasses because they centralize are too complex and centralize too much intelligence making it expensive to change them. But, what if we did not need to change them in the past?



History-based detection strategies take the evolution into account. The interesting part here is that time and structure are treated the same in the query. In Hismo, history encapsulate time and is in relation with structure. Thus, time and structure can be treated the same. This is rather philosophical :).

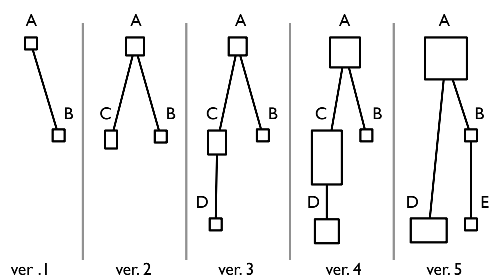
History holds useful information

When did it change?
How did it change?
What changed?
What will change?
Who did what?

How to model history?

How to model structure changes?
How to combine time with structure?
How to model changes in relationships?

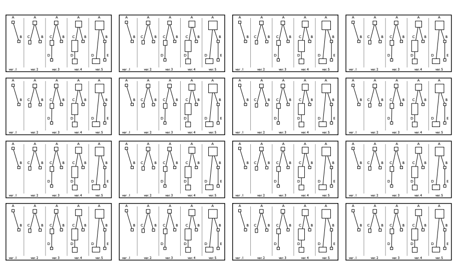
What happens with inheritance?



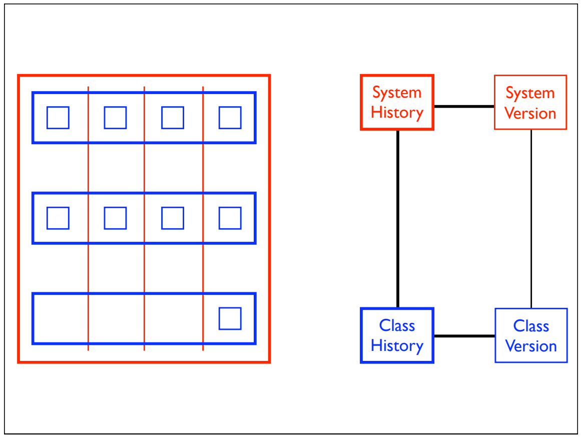
A is persistent, B is stable, C was removed, E is newborn ...

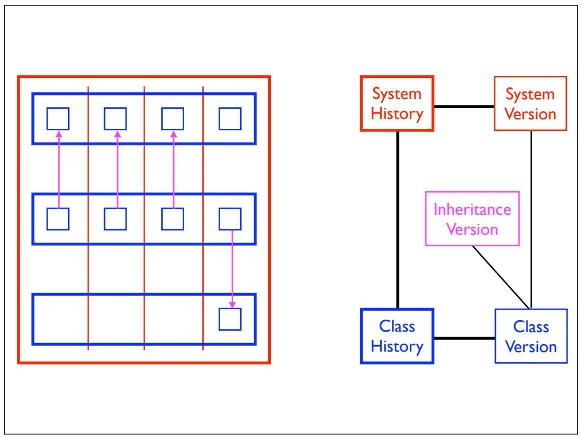
Evolution Matrix shows classes over time.
In the above picture we try to use the same approach to also show inheritance relationships.

History contains too much data

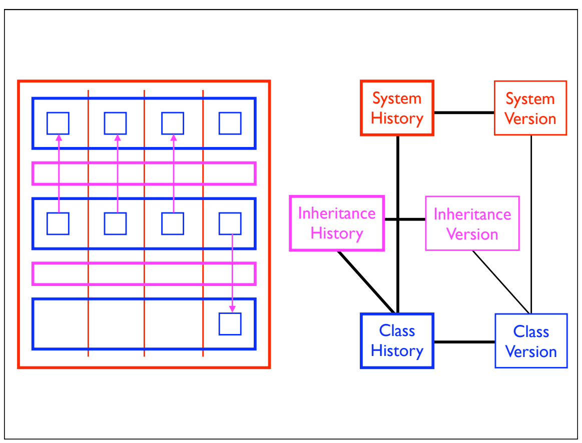


N versions means N times more data.
For several hierarchies, the approach produces unreadable pictures.

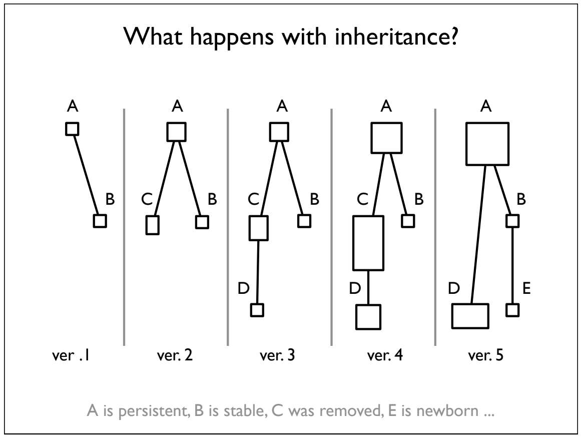




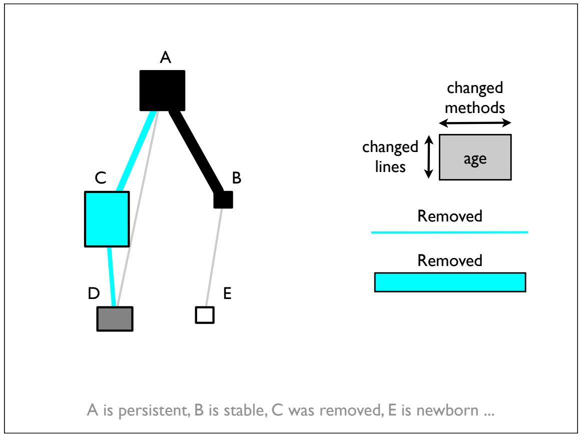
What happens if we introduce relationships? How should we model them?



Just the same as the structure. In this case, an InheritanceHistory is going to represent the historical relationship between two ClassHistories.

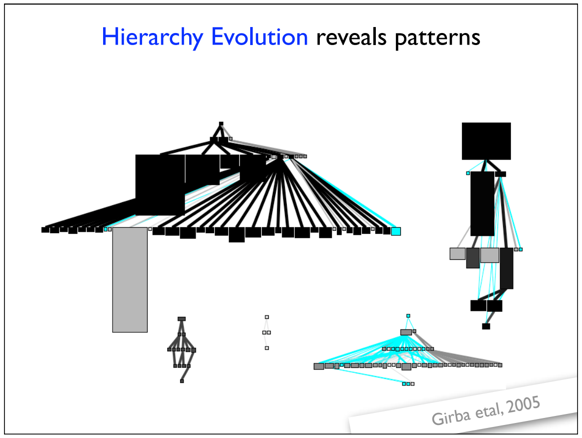


So, what can we do about this problem?



Tudor Gîrba, Michele Lanza and Stéphane Ducasse, "Characterizing the Evolution of Class Hierarchies," Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005), IEEE Computer Society, Los Alamitos CA, 2005, pp. 2—11. http://www.iam.unibe.ch/~scg/cgi-bin/scgbib.cgi/abstract=yes?Girb05a

Node = history of a class
Edge = history of the inheritance relationship between two classes
Node width = number of methods added or removed
Node height = number of statements added or removed
Node color = age (old = black, new = white)



Hierarchy Evolution View characterizes the overall activity on an entire hierarchy,

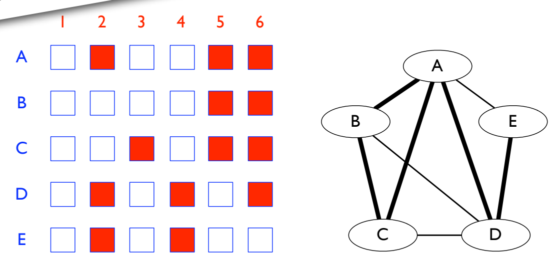
History holds useful information

When did it change?
How did it change?
What changed?
What will change?
Who did what?

How to model history?

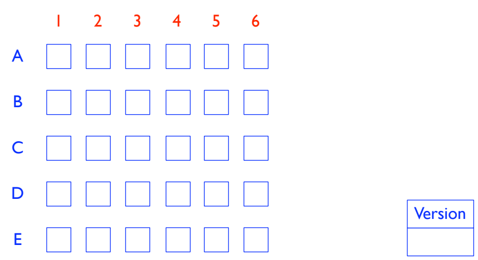
How to model structure changes?
How to combine time with structure?
How to model changes in relationships?
How to model co-changes?

Co-changes are n-ary relationships

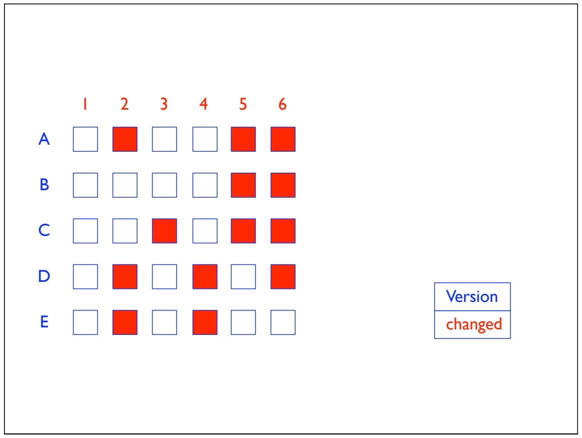


Co-changes are n-ary relationships.

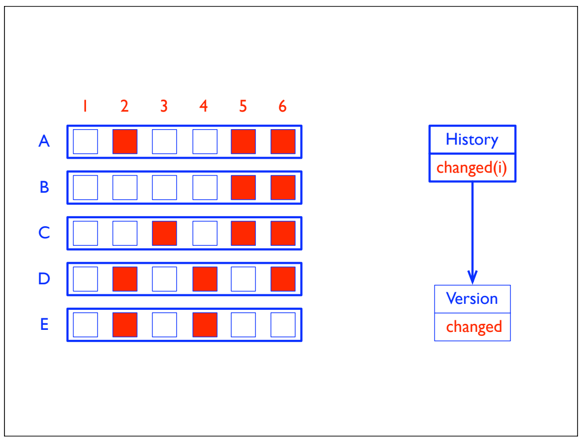
As A has a strong relationship to B, B to C and A to C, the question is if all of these are coupled due to the same reason.



How to model change?



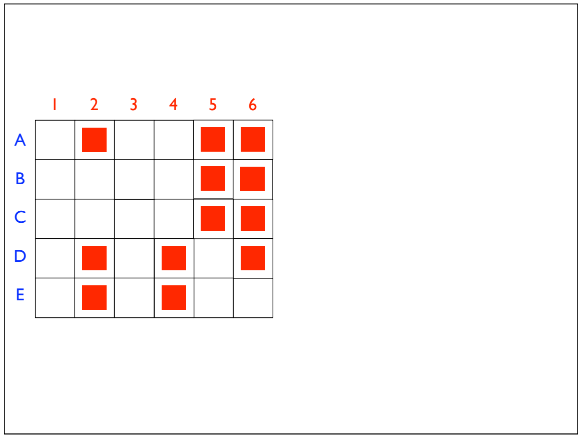
Given a Version we want to know if it changed.



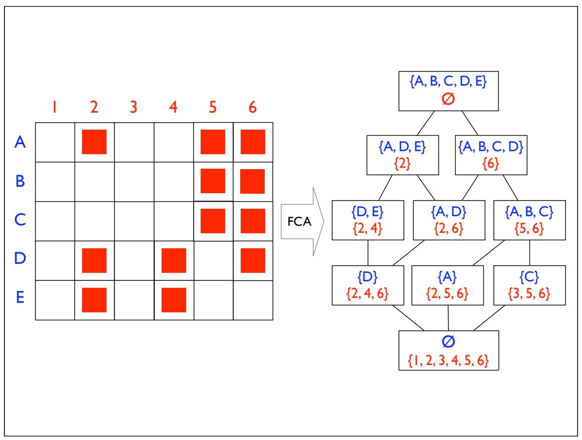
We can detect changes from a version to another, because history holds the order of versions. Thus, a History will know if it was changed in Version i.

What is Formal Concept Analysis?

We use Formal Concept Analysis to detect co-change patterns. First, let see what FCA is in a nutshell.

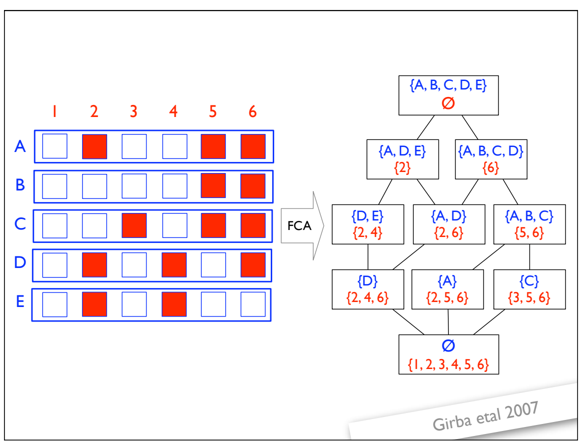


FCA takes as input a matrix of Elements (A-E) having properties (1-6).



… and offers as a result a lattice in which each node represents a concept consisting of Elements that have Properties in common. For example, A and D have 2 and 6 in common.

Coming back to our problem of detecting co-change patterns, how to we apply this technique?

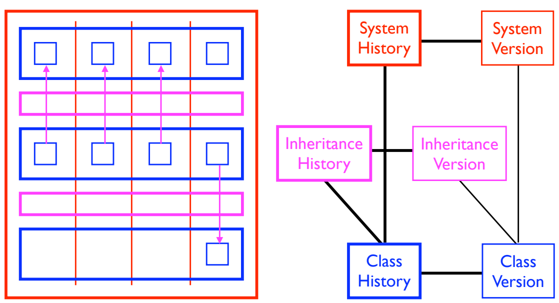


Simple. Elements are given be Histories and Properties are given by “changed in version i”. The resulting lattice shows which Histories were changed together in which versions. For example, A,B and C have changed together in 2 versions.

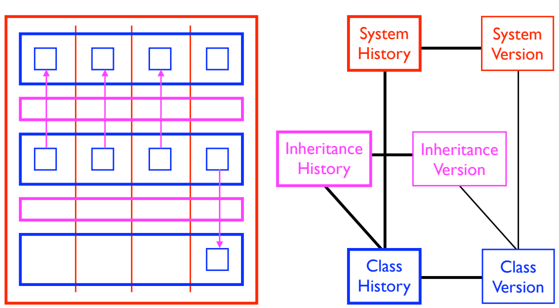
Parallel Inheritance
add simultaneously children to several classes

Shotgun Surgery
change several classes simultaneously, but do not add methods

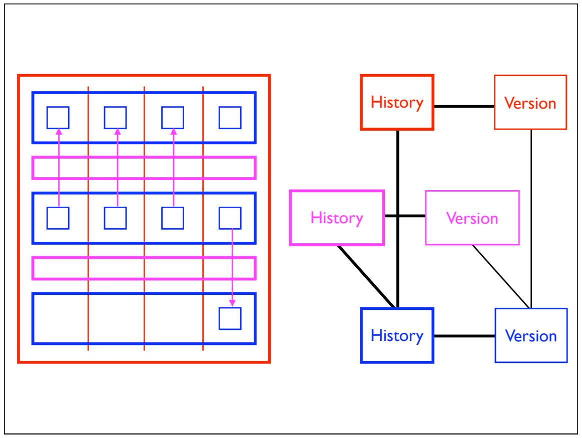
This technique can reveal different kinds of patterns depending on what Histories and what changes we take into account. For example, to detect Parallel Inheritance, we would consider ClassHistories that changed the number of children.



Hismo models history as first class entity.



Hismo models history as first class entity.



Hismo is generic. Given a structural meta-model, we can infer the historical meta-model.

History holds useful information

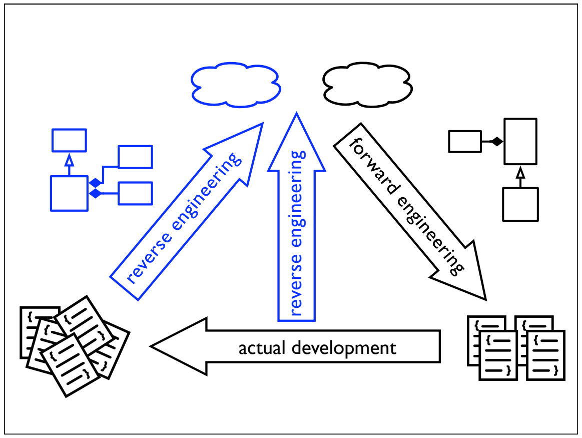
When did it change?
How did it change?
What changed?
What will change?
Who did what?

How to model history?

How to model structure changes?
How to combine time with structure?
How to model changes in relationships?
How to model co-changes?

Issues

How to sample history?
What to capture?
How to represent it?



Tudor Gîrba
www.tudorgirba.com