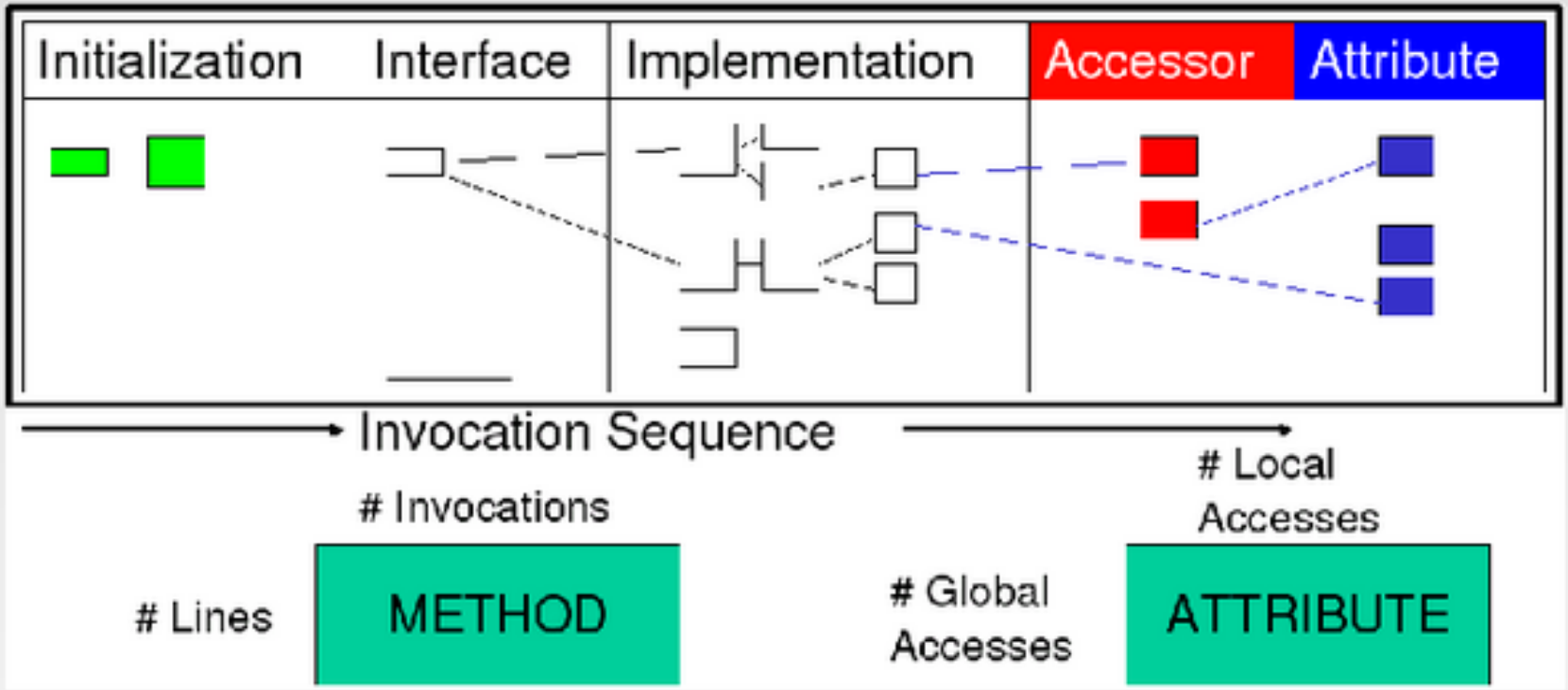
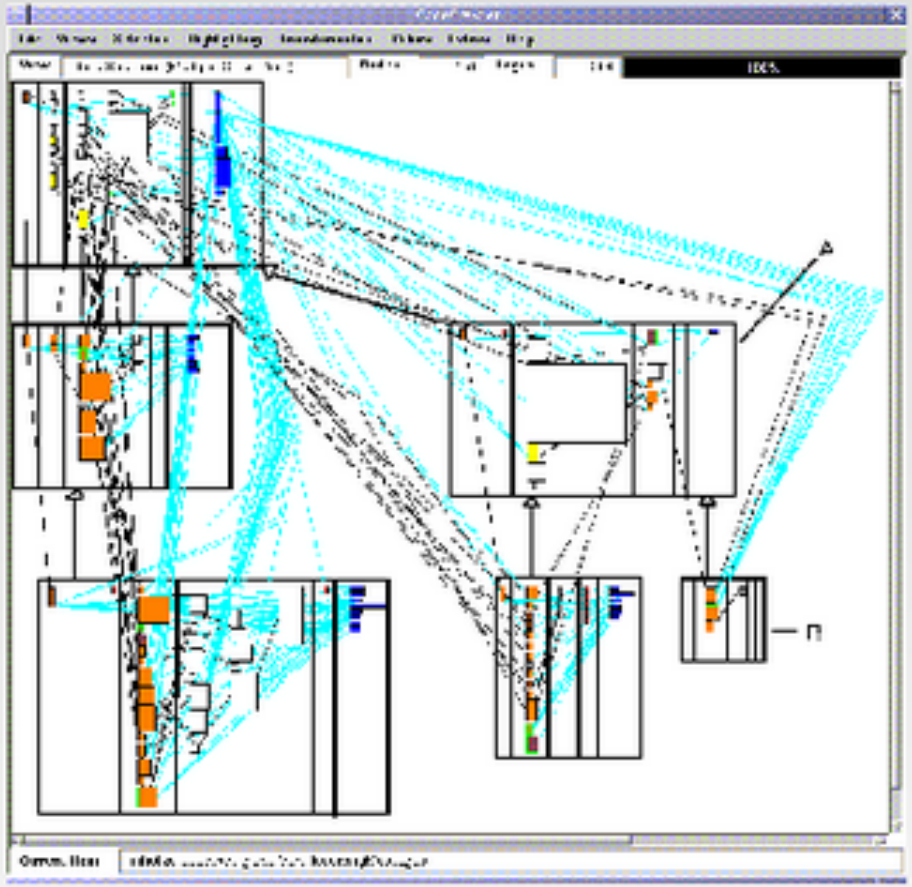
Klassenblaupause (Ducasse und Lanza 2005)



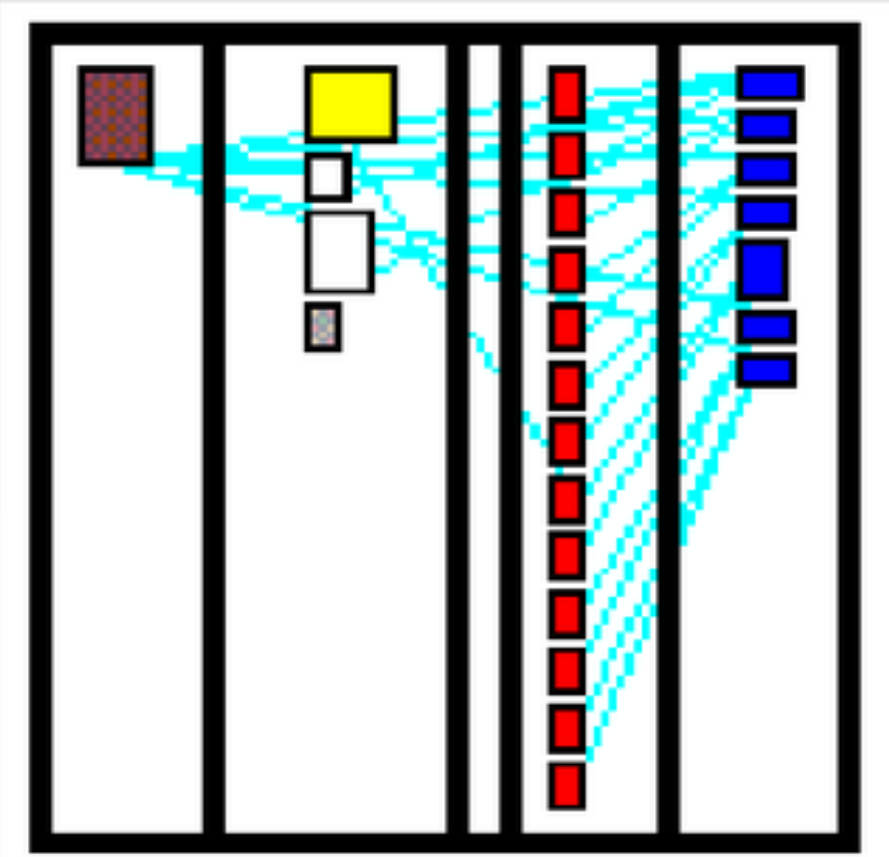
Kategorisierung von Klassen

- Basiert auf Klassenblaupausen
- Zwei Perspektiven:
  - Einzelne Klasse
  - Vererbungskontext
    - Klassenblaupause für jede einzelne Klasse
    - Sind als Baum angeordnet



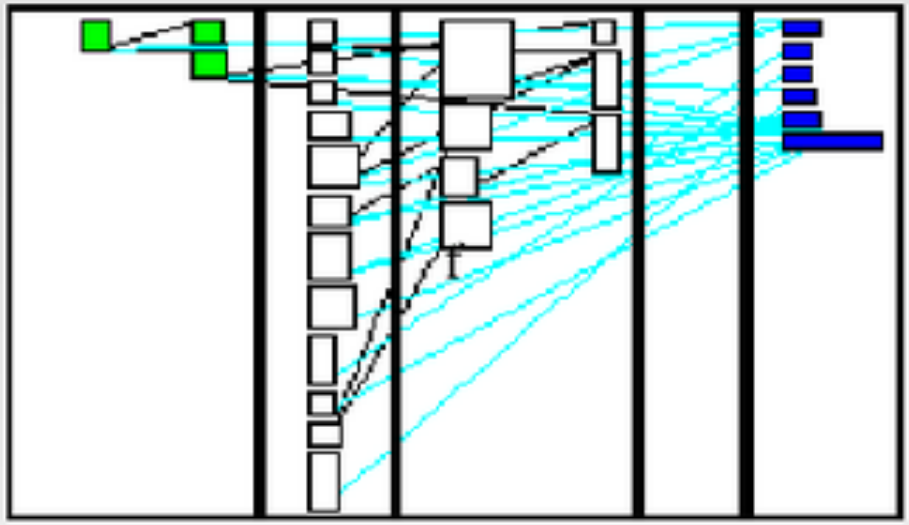
Klassenblaupause: Data Storage

- Viele Attribute
- Kann viele Zugriffsoperationen haben (Accessors)
- Harmloses Verhalten



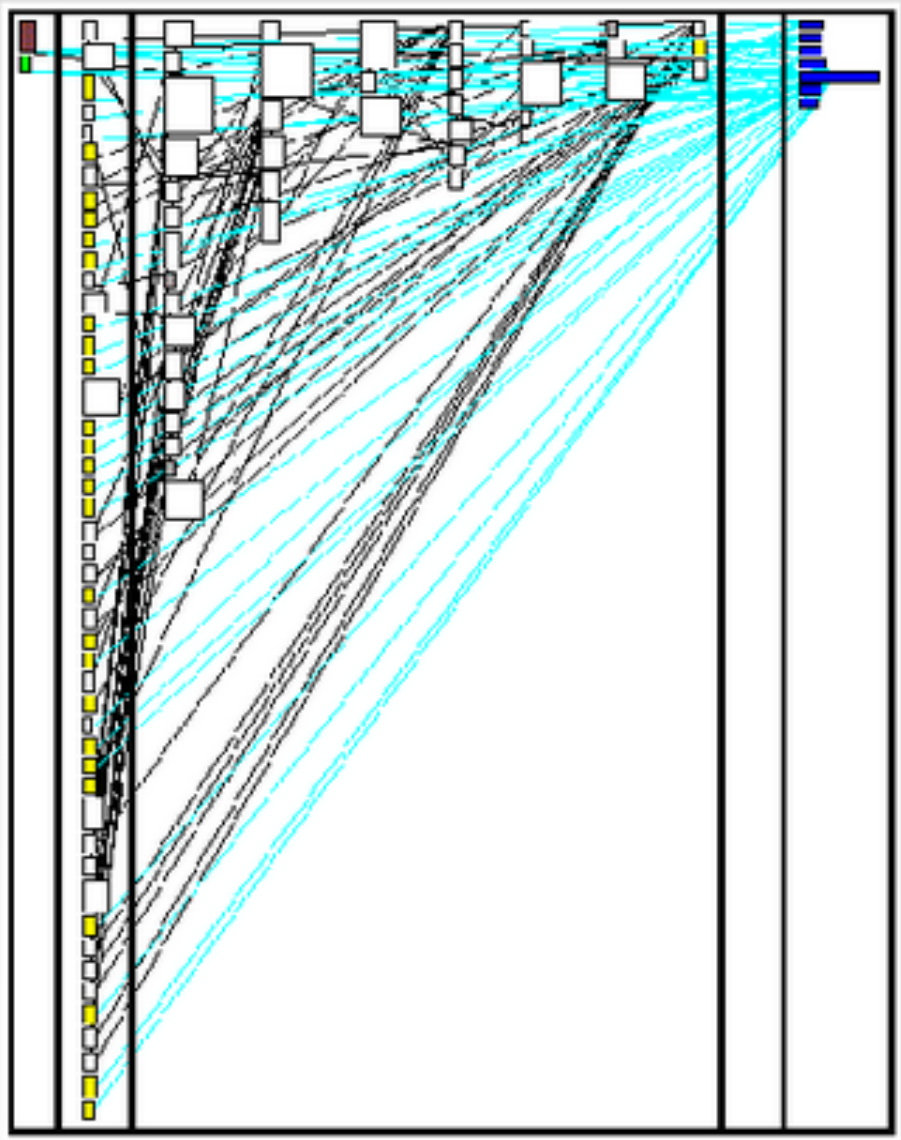
Klassenblaupause: Wide Interface

- Viele Methoden in der Schnittstelle



Klassenblaupause: Large Implementation

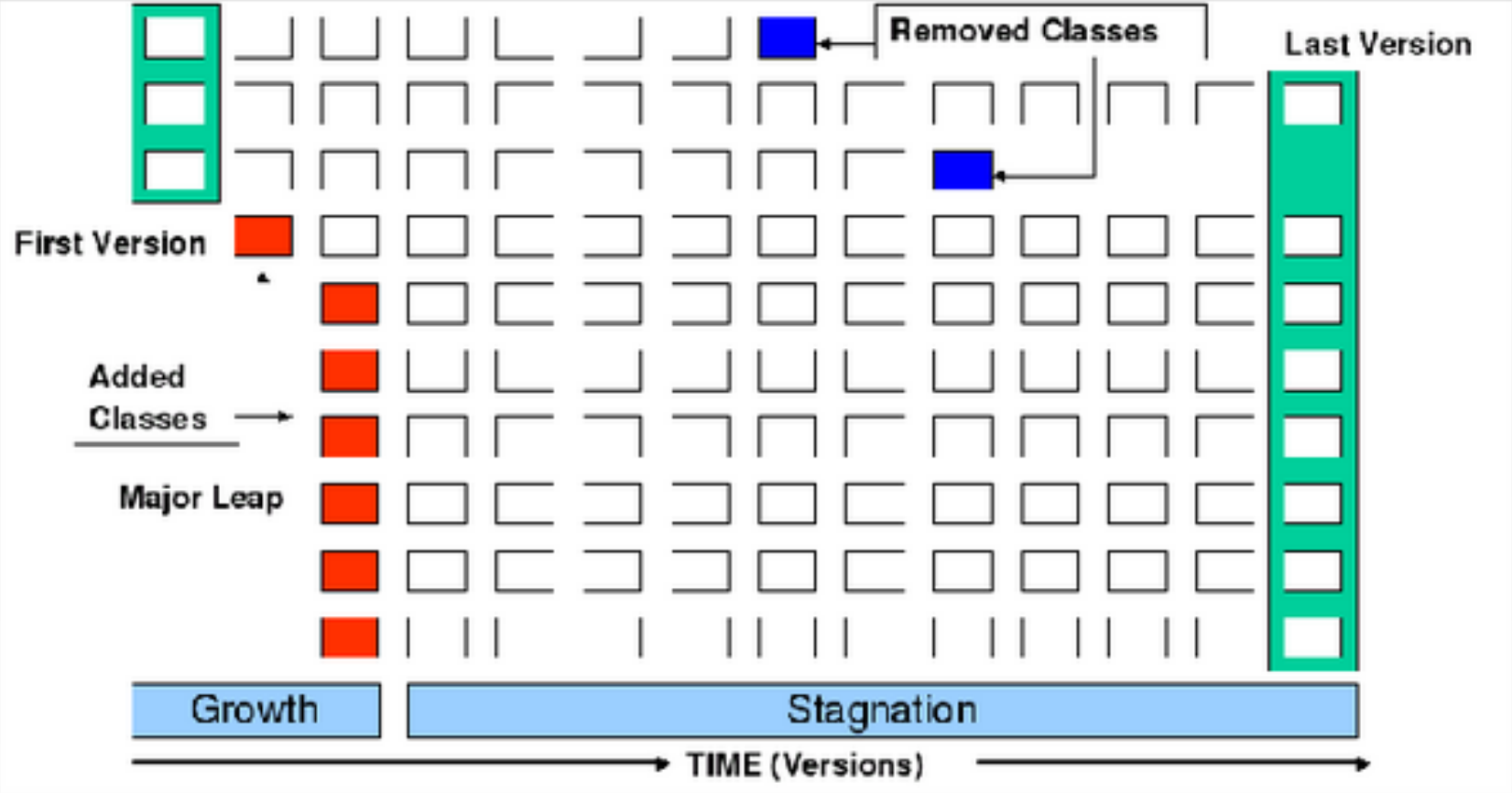
- Geschachtelte Aufrufstruktur
- Viele Methoden
- Hohe Komplexität
- Breite Schnittstelle



Evolutionäre Aspekte (Lanza 2003)

- Die Gegenwart ist oft verständlicher, wenn man die Vergangenheit kennt.
- Betrachtung von Aspekten des Systems über die Zeit.
  - Metrikwerte können besser eingeschätzt werden.
  - Erlaubt, Trends auszumachen.

Evolutionsmatrix

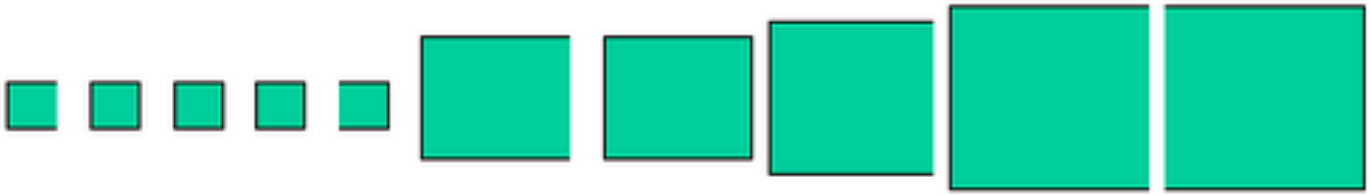


Kategorisierung von Klassen anhand der Evolutionsmatrix

- Dargestellte Metriken für Klassen:
  - NOM (number of methods)
  - NOA (number of attributes)
- Kategorisierung anhand der „individuellen Evolution“ und der „System-Evolution“:
  - Pulsar
  - Supernova
  - Weißer Zwerg
  - Roter Riese
  - Dornröschen
  - Eintagsfliege
  - Methusalem

Pulsar & Supernova

Pulsar: Wiederholte Änderungen, die sie größer und kleiner werden lassen.
System-Hotspot: Jede neue Version verlangt Anpassungen.

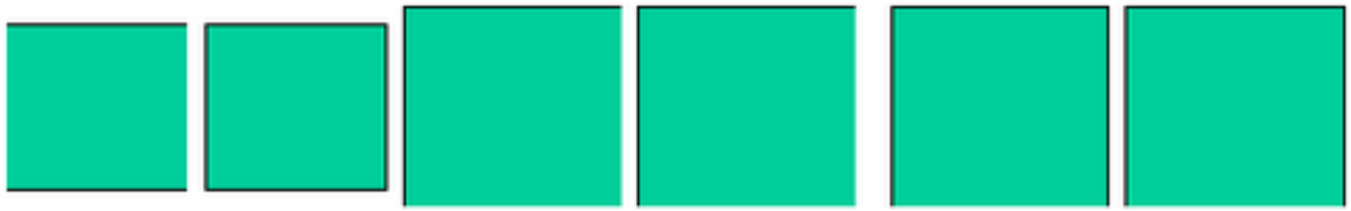


Supernova: Plötzlicher Anstieg. Mögliche Gründe:

- Massive Restrukturierung.
- Datenspeicherklasse für Daten, die plötzlich hinzukommen.
- *Schläfer:* Entwickler wußten exakt, was einzufügen ist.

Weißer Zwerg, Roter Riese, Dornröschen

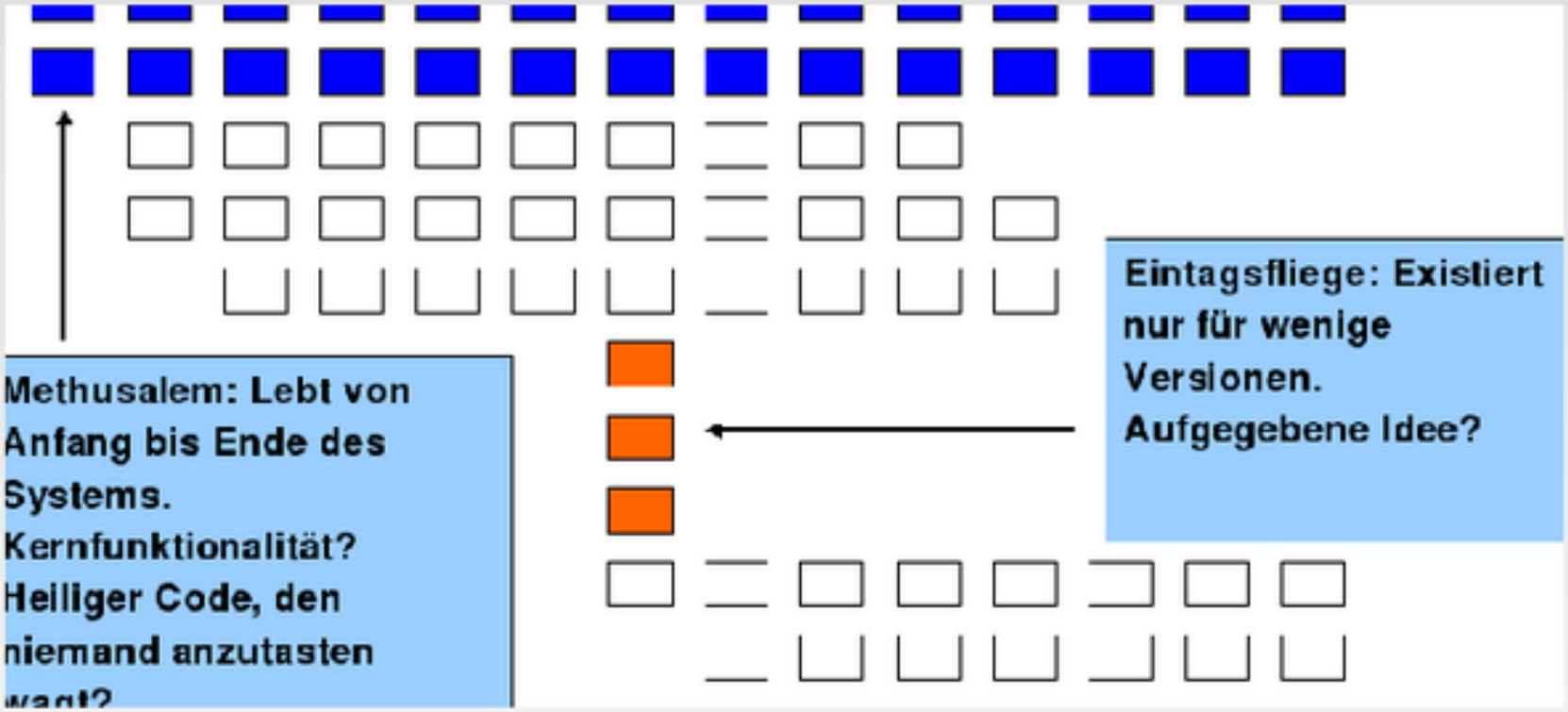
Weißer Zwerg: Verlor seine Funktionalität und dümpelt vor sich hin. Toter Code?



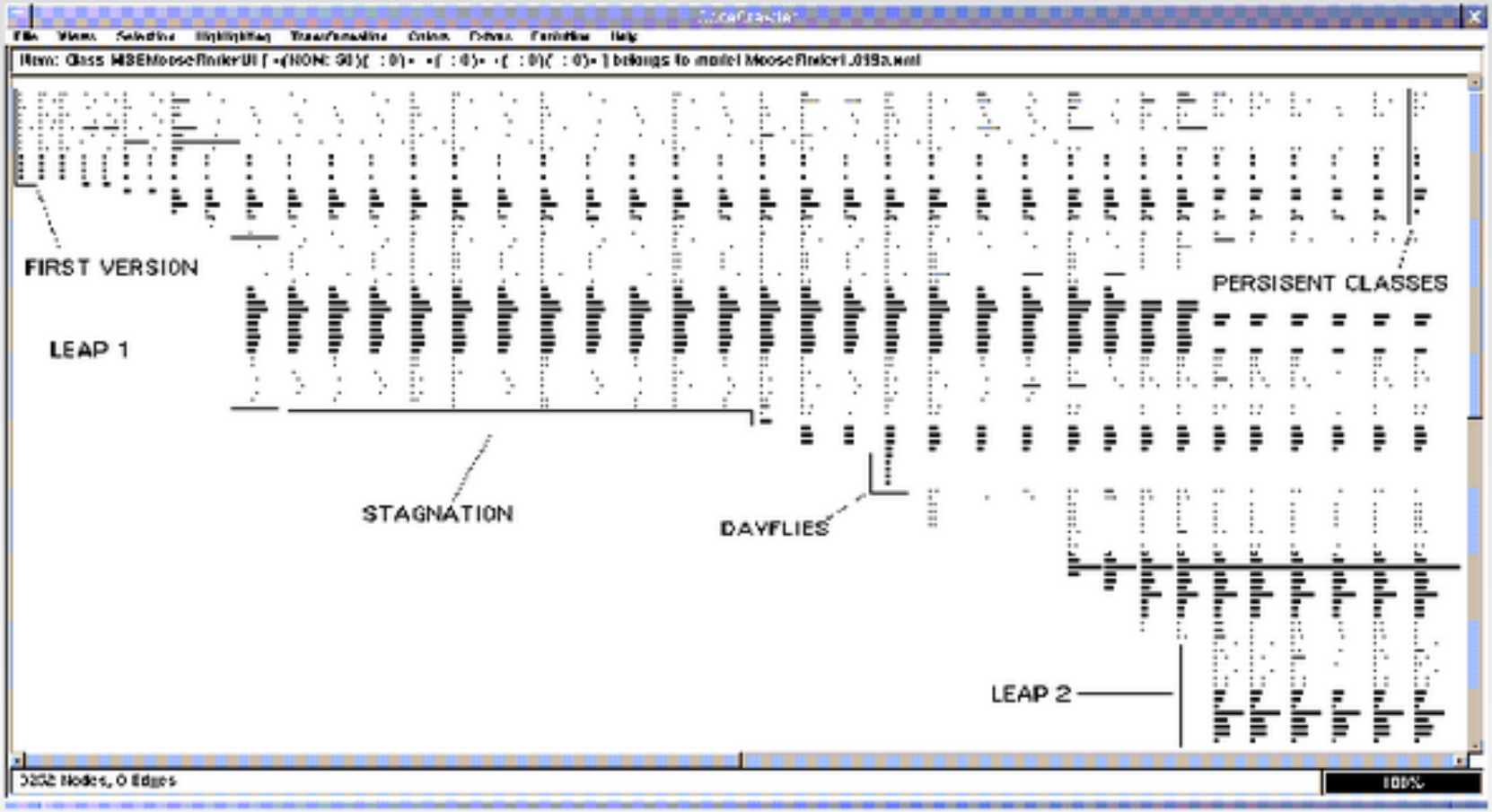
Roter Riese: Eine permanente "Gottklasse".

Dornröschen: Konstante Größe über mehrere Versionen. Toter Code? Ausgereifter Code?

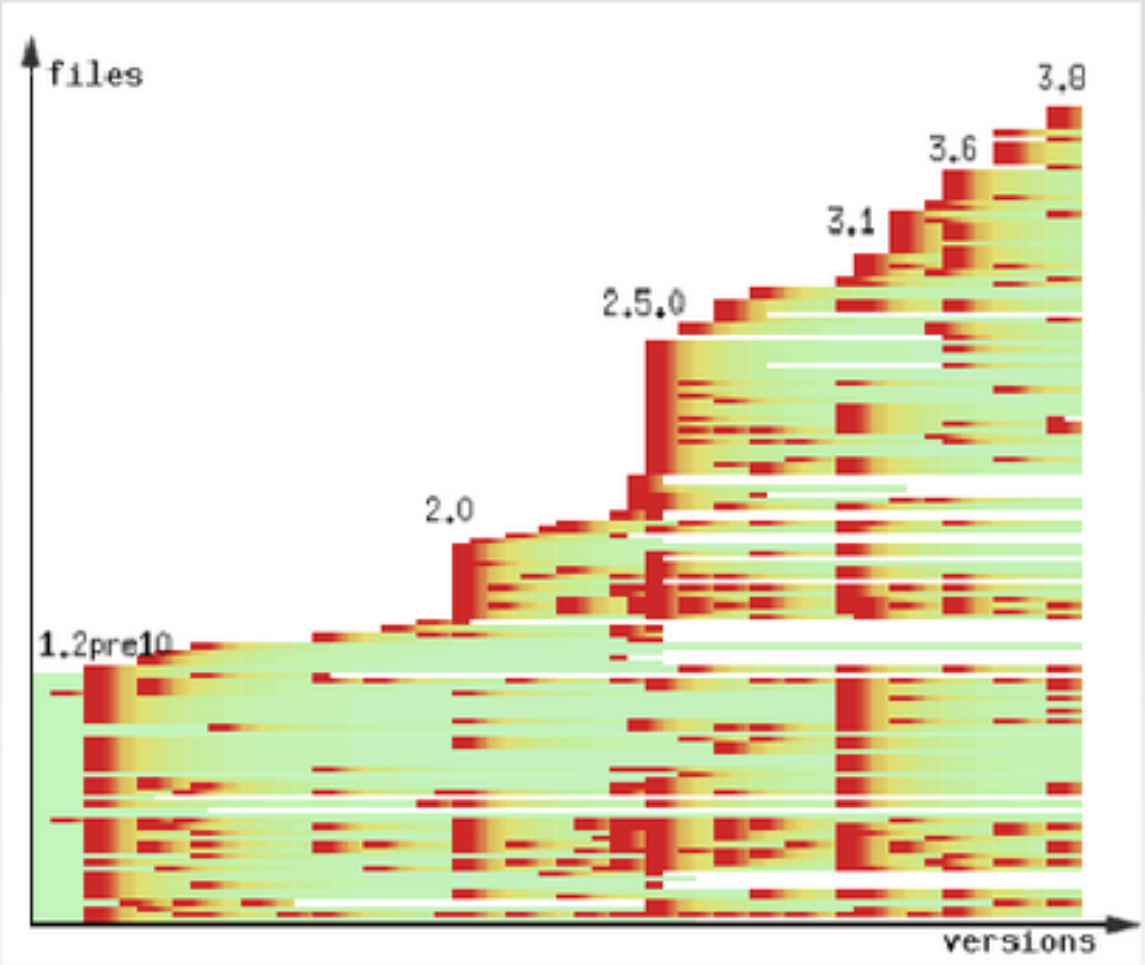
Eintagsfliege & Methusalem



Fallstudie MooseFinder (38 Versionen)



Spektrograph



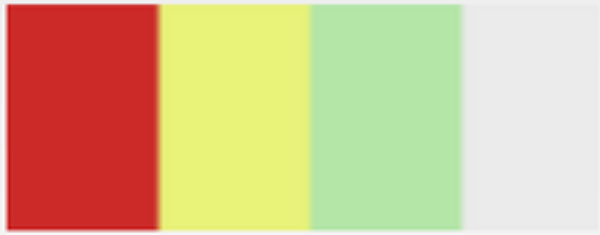
x-Achse: Zeit; y-Achse: Softwareeinheit; Farbe: #commits

Färbung im Spektrograph

- linearer Gradient

- exponentieller Gradient

- Stufen



Software-Visualisierung (SV) I

- SV ist unabdingbar im Reengineering-Kontext
- Match-Mismatch-Hypothese:

  problem-solving performance depends on whether the structure of a problem is matched by the structure of a notation

  – *Gilmore und Green*

- Jede SV betont bestimmte Information und vernachlässigt andere Information.
- Geeignete SV ist abhängig von der zu lösenden Aufgabe.

Software-Visualisierung (SV) II

- "'Alphabetismus"' der SV
  - Wie drücke ich es aus?
  - Wie interpretiere ich es?
- Vieles noch in der Forschung, wenig in kommerziellen Werkzeugen
- Software-Wahrnehmung: Andere menschliche Sinne werden genutzt

Code-Transformationen

1. Codetransformation
   - Vorteile (halb-)automatischer Transformationen
   - Definition
   - Transformationssystem
   - Schritte
   - Bestandteile
   - Implementierung
   - Eigenschaften
   - Beispiele konkreter Transformationssysteme

Code-Transformationen

- Lernziele
  - Grundsätzliche Bestandteile und Vorgehen kennen
  - Transformationen anwenden können
  - Implementierung kennen
- Kontext
  - Code-/Entwurfsebene
  - Automatische Code-Transformationen beinhalten Mustersuche
  - Aspekte des Reengineering, nicht nur des Reverse Engineerings

(Halb-)Automatische Transformationen

- (halb-)automatisch und beliebig wiederholbar
- dokumentarisch
- Transformation ist Spezifikation der Änderung
- kontrollierbar
- Transformationen repräsentieren Implementierungswissen
- Vor- und Nachbedingungen der Transformationen sind explizit; sollten automatisch prüfbar sein
- Erhalt der Semantik bei einer Transformation lässt sich u. U. zeigen

Transformation

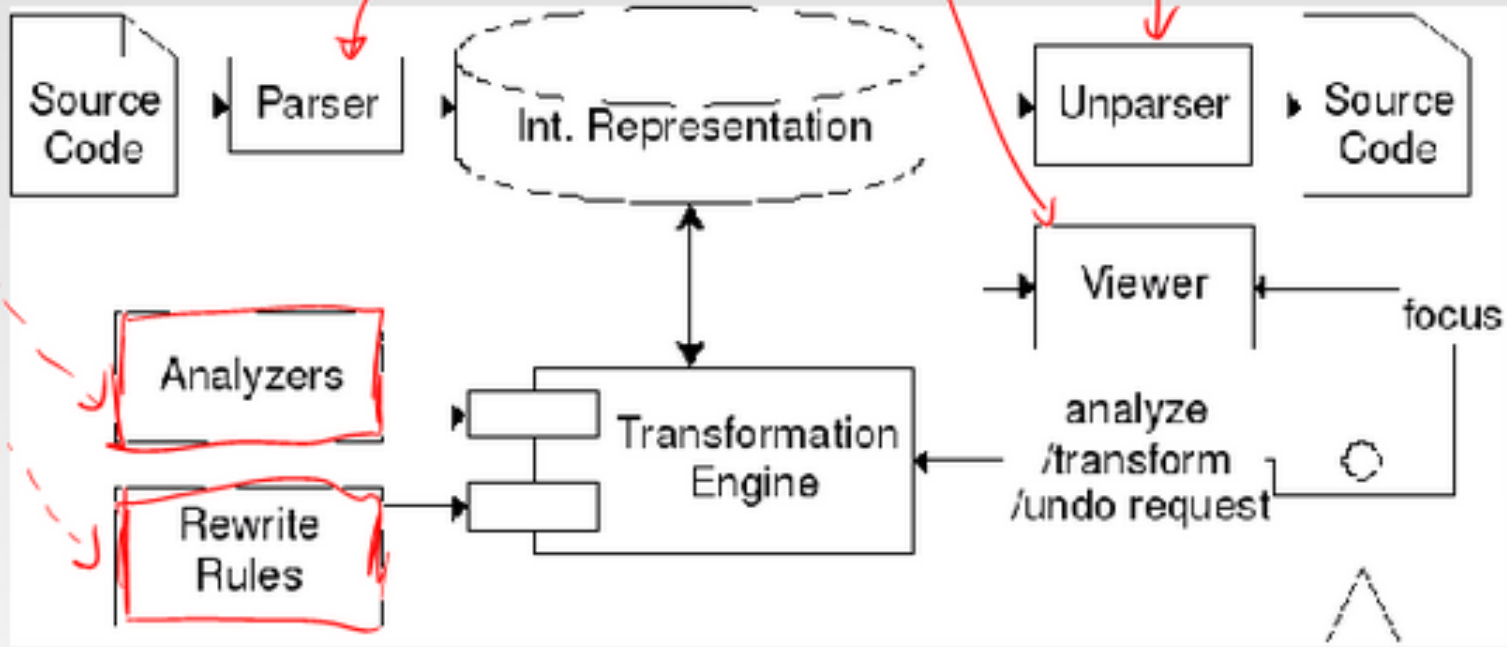
- Eine **Transformation** ist eine partielle Funktion t:
  - t: Spezifikation/Programm $\mapsto$ Spezifikation/Programm
  - Beispiele: Compiler, YACC, Programmrestrukturierer
- Transformationen werden oft als Rewrite-Regeln mit Pattern-Variablen repräsentiert:

```
rule eliminate_additive_identity
  replace [expression]
     T [expression] + 0
  by
     T
end rule
```

Syntax von TXL; siehe `www.txl.ca`

Transformationssystem

Ein Transformationssystem ist ein System, das semantisch wohl-definierte (teil-)mechanisierte Programmmodifikationen ermöglicht.



Schritte einer Transformation

1. Lokation: Identifikation der Stelle, an der Transformation angewandt werden soll

Vorbedingungen von Transformationen

Die Anwendbarkeit von Transformationen ist oft an Bedingungen geknüpft:

```
rule eliminate_additive_identity
    replace [expression]
        T1 [expression] + T2 [expression]
    where
        value (T2) = 0
    by
        T1
    end rule
```

Umsetzung von Transformationen

Ersetzung wird als Prozedur auf dem abstrakten Syntaxbaum implementiert:

```
procedure eliminate_additive_identity (n : in out node) is
begin
  if type (n) = plus then    -- Mustersuche
    if value (n.right) = 0 -- Bedingung
      then
                           -- Ersetzung
      replace_child (parent (n), from => n, to => n.left);
      n.left.parent := n.parent;
      delete_tree (n.right); delete (n);
    end if;
  end if;
end;
```