# 30. Refactoring based on Metaprogramming

Andreas Ludwig

Prof. U. Aßmann

http://recoder.sf.net

# Overview

❶ Programming in the Large and Refactoring

- Problems, Concepts, The Approach

❷ The Architecture of RECODER

- Requirements, Separation of concerns, Dataflow, Models, Algorithms

❸ Generic Refactoring Systems

- Abstract Requirements

# Obligatory Literature

- Tom Mens and Tom Tourwe. A survey of software refactoring. IEEE Transactions on Software Engineering, 30, 2004.
- http://informatique.umons.ac.be/genlog/resources/refactoringPapers.html
- Ludwig, Andreas and Heuzeroth, Dirk. Meta-Programming in the Large, Generative Component-based Software Engineering (GCSE), ed. Eisenecker, U. W. and Czarnecki, K., Erfurt, Germany, pages 443-452, Springer, Lecture Notes in Computer Science 2177, 2001

http://dx.doi.org/10.1007/3-540-44815-2_13

http://www.springerlink.com/content/f56841633653q258/

# Non-Obligatory Literature

- James O. Coplien, Liping Zhao. Symmetry Breaking in Software Patterns. Springer Lecture Notes in Computer Science, LNCS 2177, October 2001, ff. 37. http://users.rcn.com/jcoplien/Patterns/Symmetry/Springer/SpringerSymmetry.html
- W. Zimmer. Frameworks und Entwurfsmuster. Dissertation, Universität Karlsruhe, 1997, Shaker-Verlag.
  - Benedikt Schulz, Thomas Genssler, Berthold Mohr, Walter Zimmer. On the Computer-Aided Introduction of Design Patterns into Object-Oriented Systems. Proceedings of TOOLS 27 -- Technology of Object-Oriented Languages and Systems, J. Chen, M. Li, C. Mingins, B. Meyer, 1998. The first time, refactorings were automatied in a CASE tool (Together)

# Refactoring

A **refactoring** is
a semantics-preserving, but structure-changing transformation of a program.

Often, the goal is a design pattern.

# A Little History

- 80s: Broad-spectrum languages (CIP)
- System REFINE
- 1992 William Opdyke coined the term *refactoring*
- 1997, Karlsruhe University started a refactoring tool
  - Based on Walter Zimmer's thesis “Design patterns as operators”
  - Idea: a refactoring is a *semantics preserving* ***operator***, transforming class graphs to class graphs
  - A refactoring operator can be implemented as a static metaprogram
- 1998, during Zimmer's work was reimplemented into the Together CASE tool, the world-wide first CASE tool with refactoring support

# Classes of Refactorings

- **Rename Entity**
  - Problem: update all references on definition-use-graph
- **Move Entity**
  - Move class feature (attribute, method, exception,...)
  - Problem: shadowing of features along scoping
- **Split Entity** or **Join Entity**
  - Method, class, package
  - Problem: updating of references
- **Outline Entity** (Split Off) or **Inline Entity** (Merge)
  - Method, generic class
  - Problem: introduction of parameters

# Steps of a Refactoring

- [Mens/Tourwe]

1) Find the place
2) Select the appropriate refactoring
3) Analyze and verify that the refactoring does not change semantics
4) Do it
5) Reanalyze software with regard to qualities such as structure, performance, etc.
6) Maintain consistency of software with secondary artefacts (documentation, test suites, requirement and design specifications etc)

# Example: Rename Refactorings in Programs

How to change the name of variable Foo and keep the program consistent?

Refactor the name `Person` to `Human`:

```
class Person { .. }
class Course {
   Person teacher = new Person("Jim");
   Person student = new Person("John");
}
```

Definition

Reference (Use)

```
class Human { .. }
class Course {
   Human teacher = new Human("Jim");
   Human student = new Human("John");
}
```

# Definition-Use Graphs (Def-Use Graphs) as a Basis of Refactorings

- Every language and notation has
  - Definitions of items (define the variable Foo)
  - Uses of items (references to Foo)
- This is because we talk in specifications about *names of objects* and their use
  - Definitions are done in a data definition language (DDL)
  - Uses are part of a data manipulation language (DML)
- Starting from the abstract syntax, the *name analysis* finds out about the definitions, uses, and their relations (the *Def-Use graph*)
  - Def-Use graphs exist in every language!
  - How to specify the name analysis, i.e., the def-use graph?

# Refactoring on Def-Use Graphs

- For renaming of a definition, all uses have to be changed, too
  - We need to trace all uses of a definition in the Def-Use-graph
  - Refactoring works always on Def-Use-graphs
- Refactoring works always in the same way:
  - Change a definition
  - Find all dependent references
  - Change them
  - Recurse handling other dependent definitions
- Refactoring can be supported by tools
  - The Def-Use-graph forms the basis of refactoring tools
- However, building the Def-Use-Graph for a complete program costs a lot of space and is a difficult program analysis task
  - Every method that structures the Def-Use-Graph benefits immediately the refactoring
  - either simplifying or accelerating it

# Programming in the Large (1)

How to organize and maintain systems with thousands of components?

- Software development becomes more than Algorithms & Data Structures.
  - Interface design is a global optimization problem
  - Many non-functional, often contradicting criteria such as efficiency, interface complexity, robustness, flexibility, ...
- There are non-local dependencies: Changes concerning interfaces and relations become a risk.
  - Hard to foresee what further changes will emerge.
  - Risks: Delay, failure, new bugs...

# Programming in the Large (2)

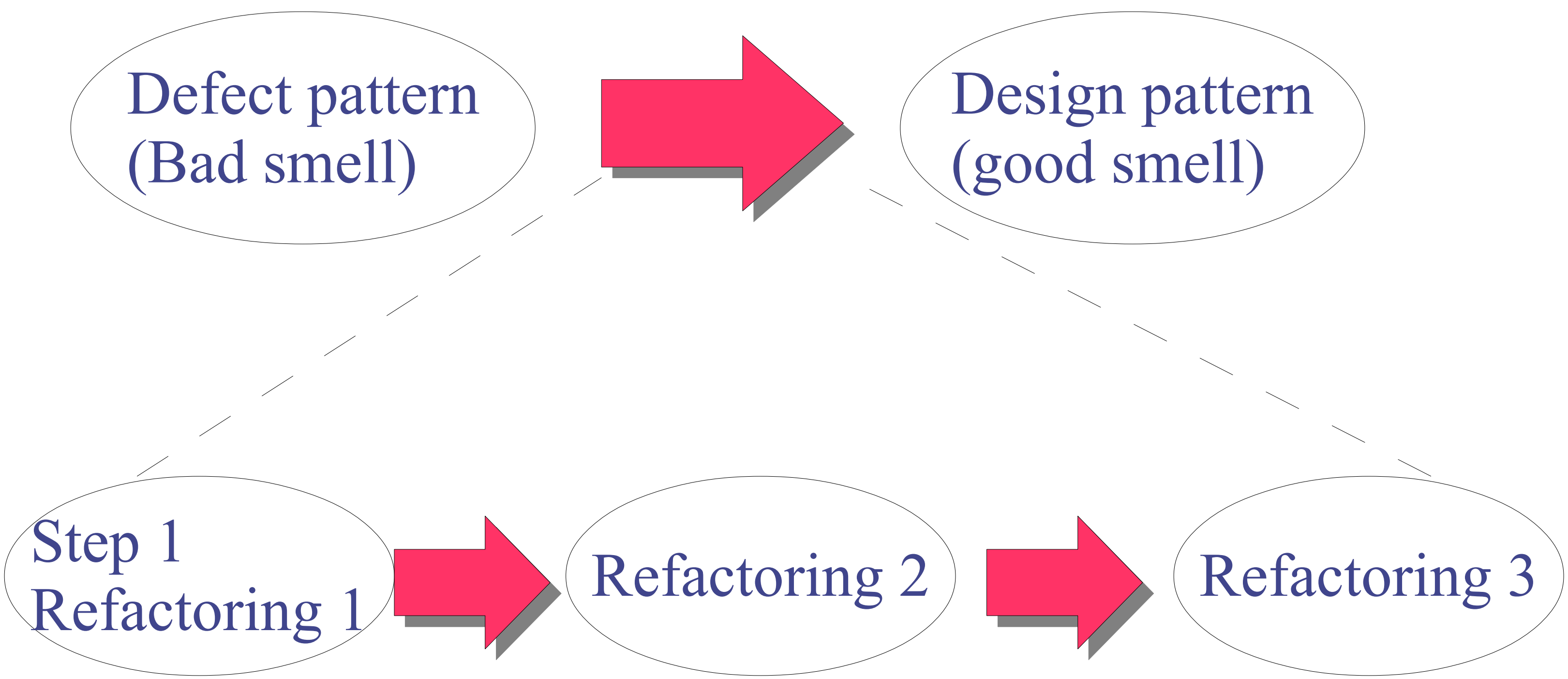
- What does *change* mean?
  - Reconfiguration: Replace old solutions
    - Variability and extensibility
  - Adaptation: Migrate to new interfaces
    - Reengineering: Problem detection comes first
  - Evolution: Improve the program iteratively and incrementally.
- An ideal developer would never have to touch his interfaces.

# Programming in the Large (3)

- Idea 1: Use development concepts that allow to express changes locally.
- Idea 2: Apply brute force and change globally regardless of costs.
  - Employ 1000 programmers?
  - Run a program?
- Idea 3: Automate the process of introducing design patterns.

# Refactorings Transform Antipatterns Into Design Patterns

- A DP can be a goal of a refactoring



# The Metaprogramming Approach to Refactoring

- Program sources are formal languages and contain a lot of accessible information.
  - We can analyze and transform programs, especially interface related code (“glue”).
- A program manipulates data.
- A metaprogram is a program that manipulates programs.
- A metaprogram is a partial compiler.
  - Source-to-source?
  - At compile time?
  - Used iteratively for incremental changes?

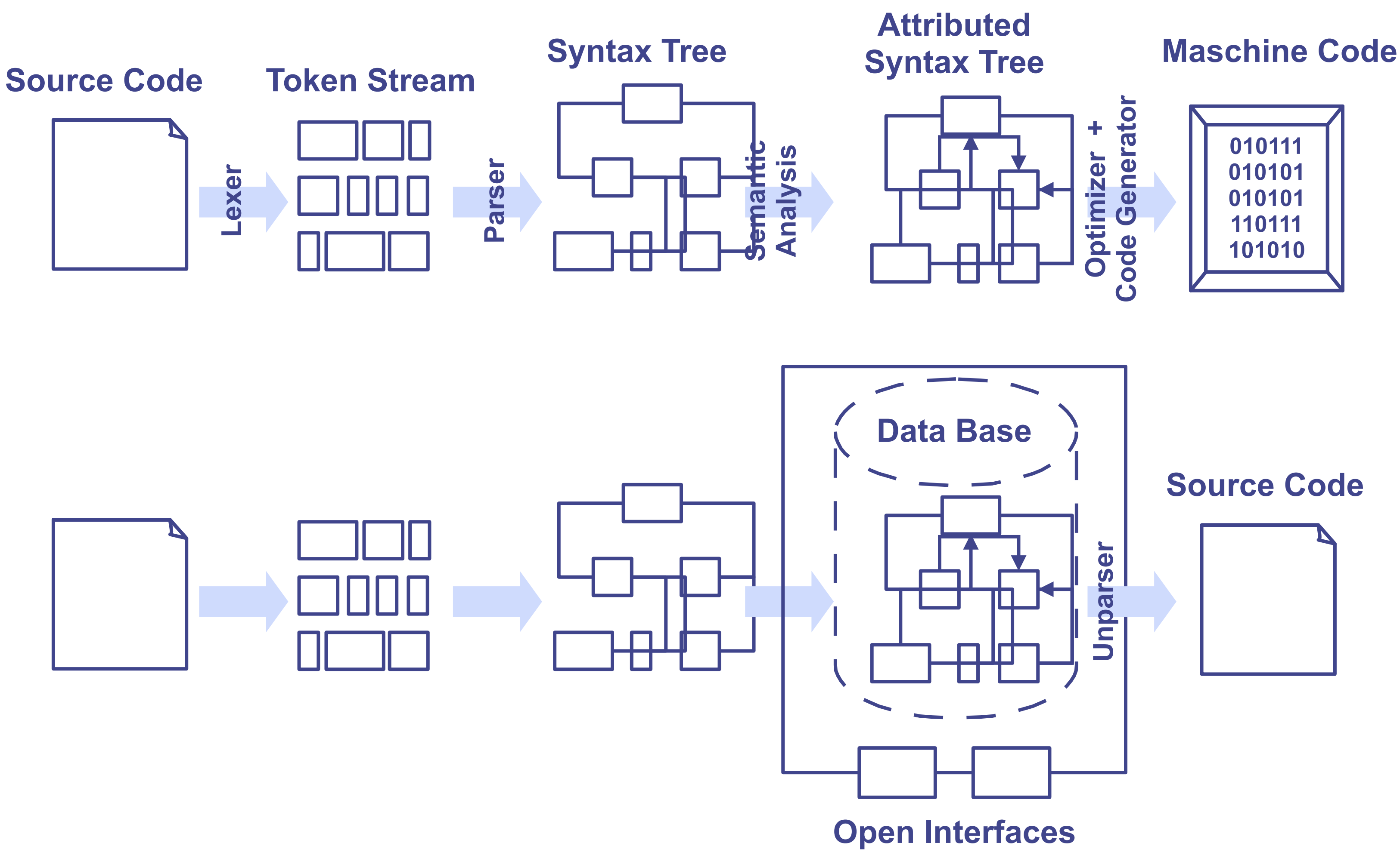
# Metaprogramming Variants

| Languages \ Times | Static<br>Compile / Link | Dynamic<br>Load / Run |
|---|---|---|
| S → S<br>**Code Structuring**<br>**Incrementality** | **Program Transformations, Pattern Refactorers** | Reflexive Program |
| S → S'<br>**Code Extension** | **Preprocessor, Code Generator, Aspect Weaver** | |
| S → B | Compiler | Just-In-Time Compiler |
| B → S<br>**Code Formatting** | Decompiler | |
| B → B<br>**Incrementality** | Binary Code Optimizer, Linker | Loader, Run Time Optimizer |
| B → B' | Binary Code Cross Compiler | Emulator |

# Refactoring Engine RECODER

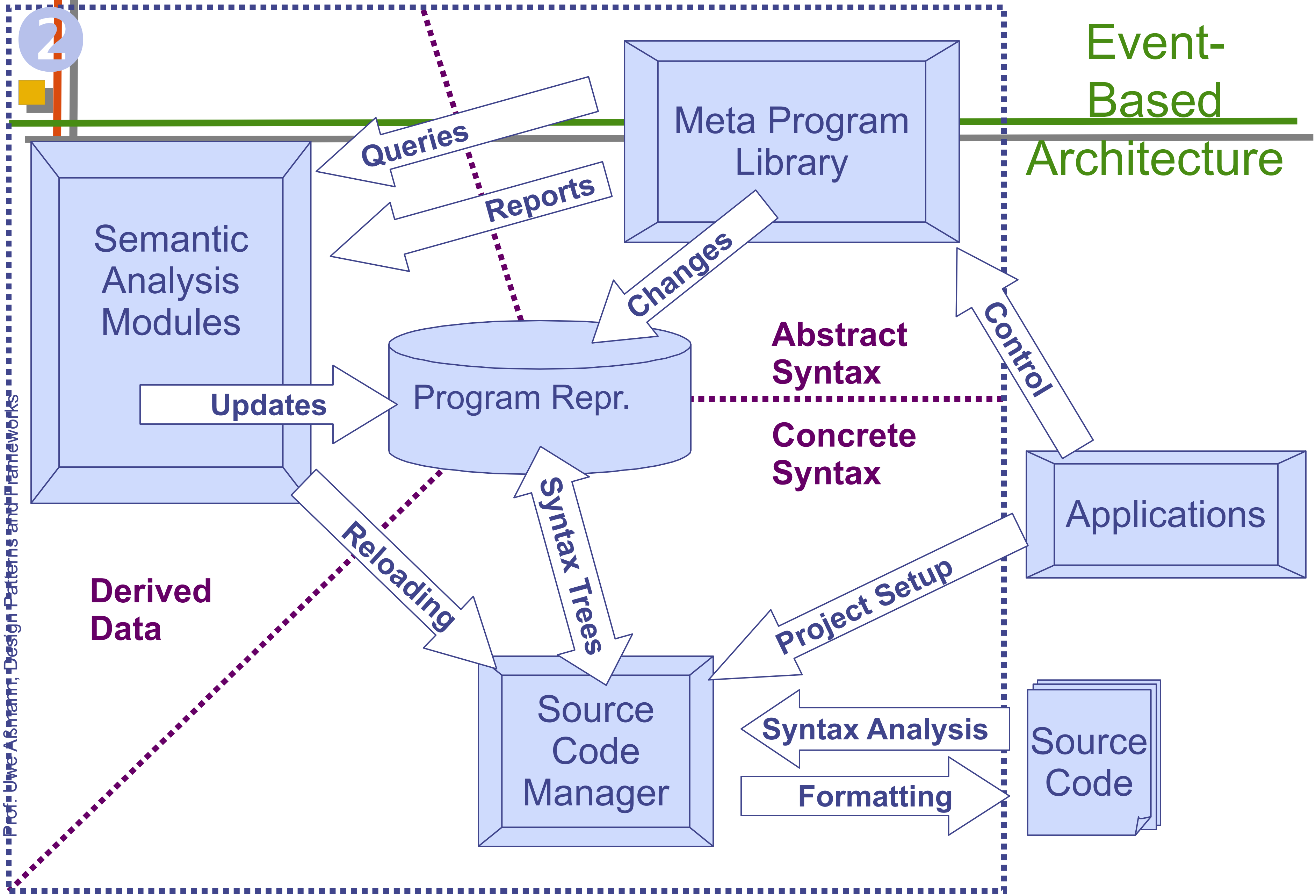
- Contains a compiler-like front-end and a source-to-source transformation library (metaprograms)
- ≈ 100000 LOC (core: ≈ 75000 LOC)
- ≈ 650 classes (core: ≈ 500 classes)
- 5 person-years development.
- Supports Java, including nested classes.

# Compiler versus Source Transformation System



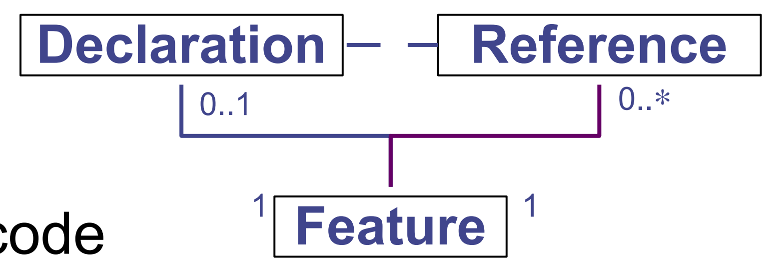
# Design Requirements for Refactoring Tools

- Easy to use refactoring-API
  - Split functionality into services.
- Deal with any query at any time: Lazy evaluation.
- Retain Source Structure (source code hygenic)
  - Model must contain structural information.
- Incremental Evaluation
  - Keep cached data consistent, efficiently
  - Incremental Analysis


Event-Based Architecture
Meta Program Library
Queries
Reports
Semantic Analysis Modules
Changes
Control
Abstract Syntax
Program Repr.
Updates
Concrete Syntax
Applications
Reloading
Syntax Trees
Derived Data
Project Setup
Source Code Manager
Syntax Analysis
Formatting
Source Code

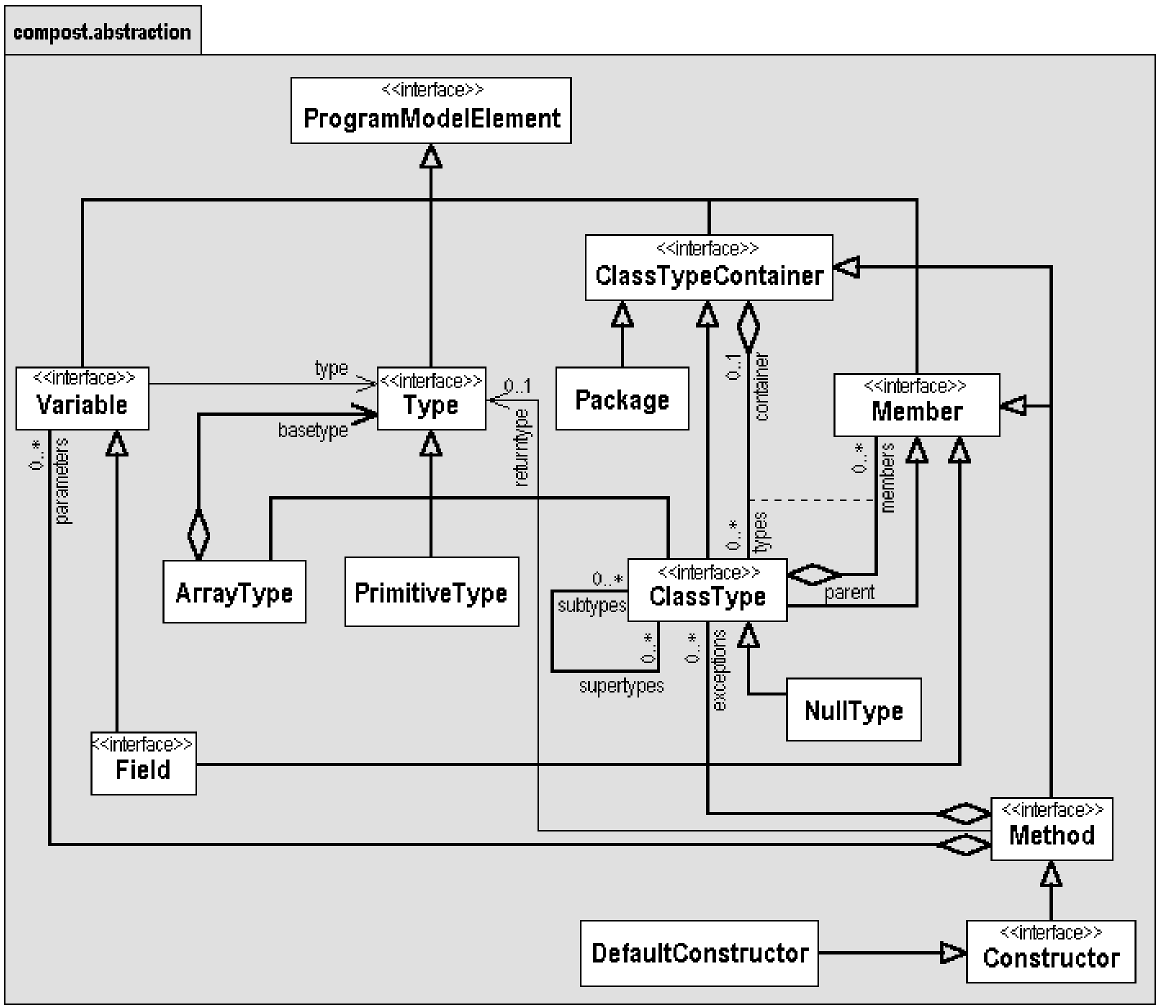
# RECODER Java Model

- Java attributed syntax graph (ASG)
- Parent links for efficient upward navigation in the scopse
  - Linking and unlinking must be done consistently.
- Abstract supertypes
  - Containment properties
  - Scoping properties
  - Commonalities with byte code

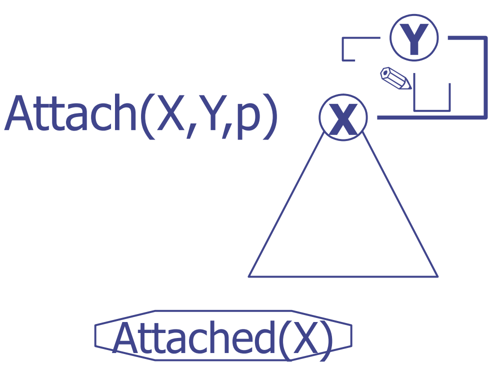



- Bidirectional definition-reference relation (name resolution + cross referencing)

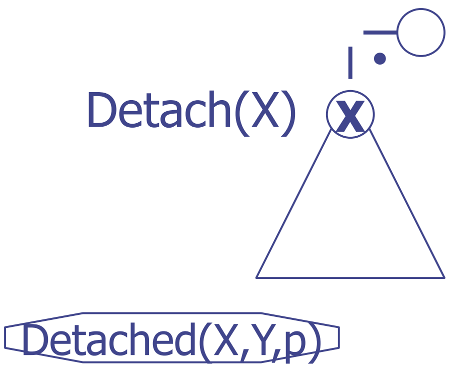
# Abstract Java Program Metamodel



# Event-based Architecture: Changes and Change Events in a Refactorer

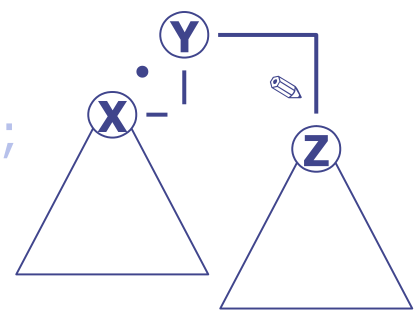



Define changes in terms of atomic Transformations




Reduce all complex changes to atomic ones.

```
Replace(X,Z) {
    Y = Parent(X);
    p = Position(X,Y);
    Detach(X);
    Attach(Z,Y,p);
}
```

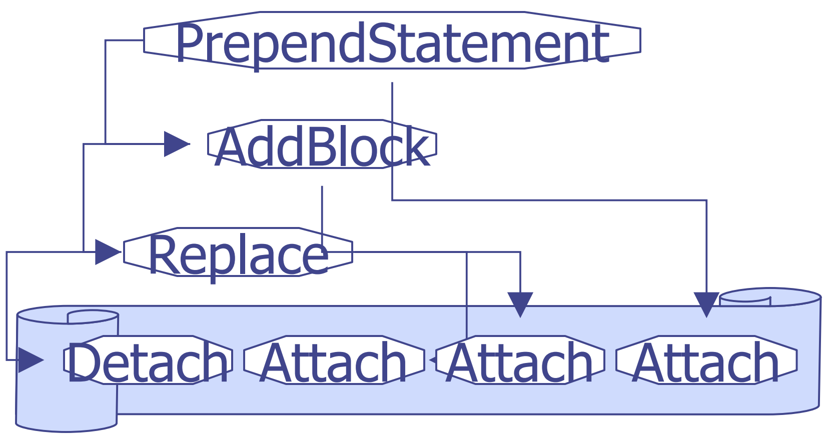



Replaced(X,Y)

# Example Change Report

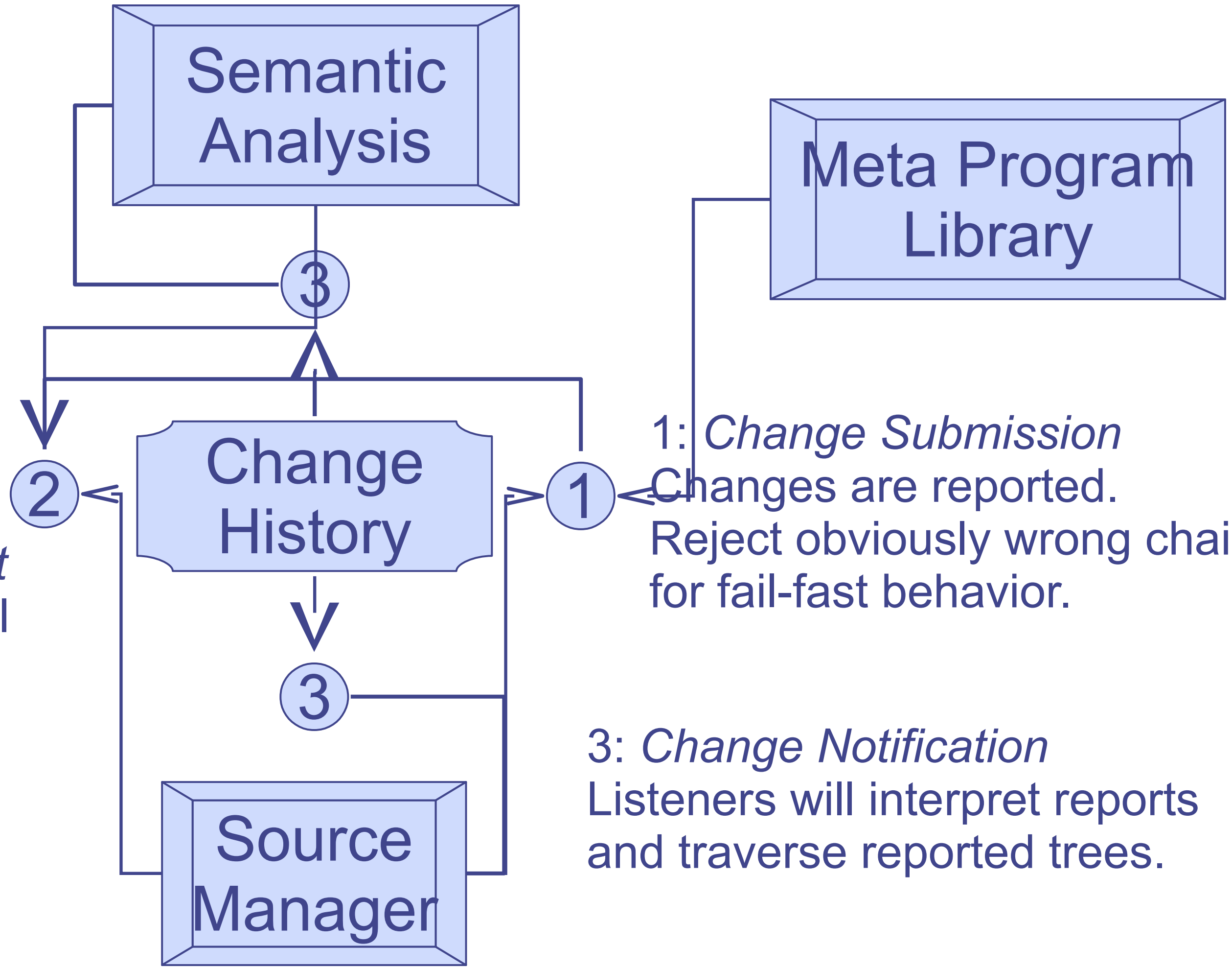
```
if (expr) stmtS;
```

```
if (expr) {
  stmtR;
  stmtS;
}
```




```
PrependStatement(R, S) {
    B = Parent(S)
    if B is no Block {
        B = AddBlock(S);
        p = 0;
    } else {
        p = Position(S)
    }
    Attach(R, B, p);
}
AddBlock(S) {
    B = new Block;
    Replace(S, B);
    S' = CloneTree(S);
    Attach(S', B, 0);
    return B
}
```

# Change Report Propagation



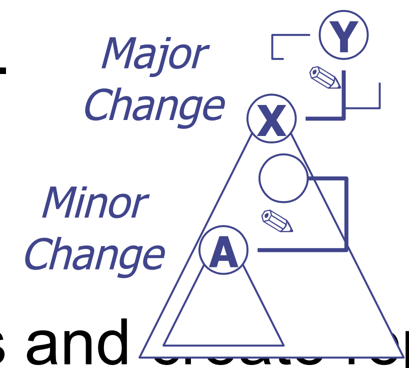

1: *Change Submission*
Changes are reported.
Reject obviously wrong chains for fail-fast behavior.

2: *Update Request*
Enforce that model is up to date now.

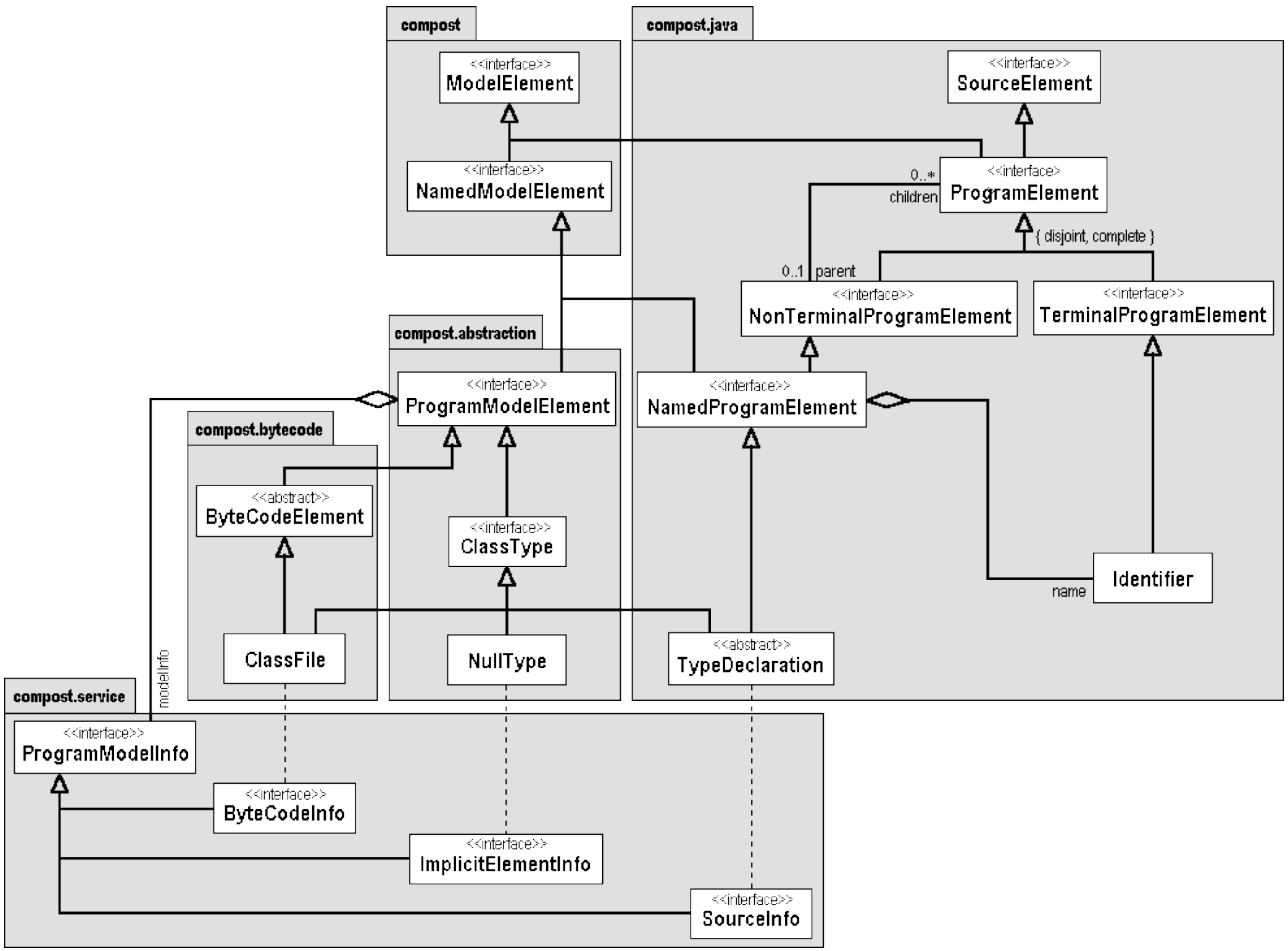
3: *Change Notification*
Listeners will interpret reports and traverse reported trees.

# Change Report Handling

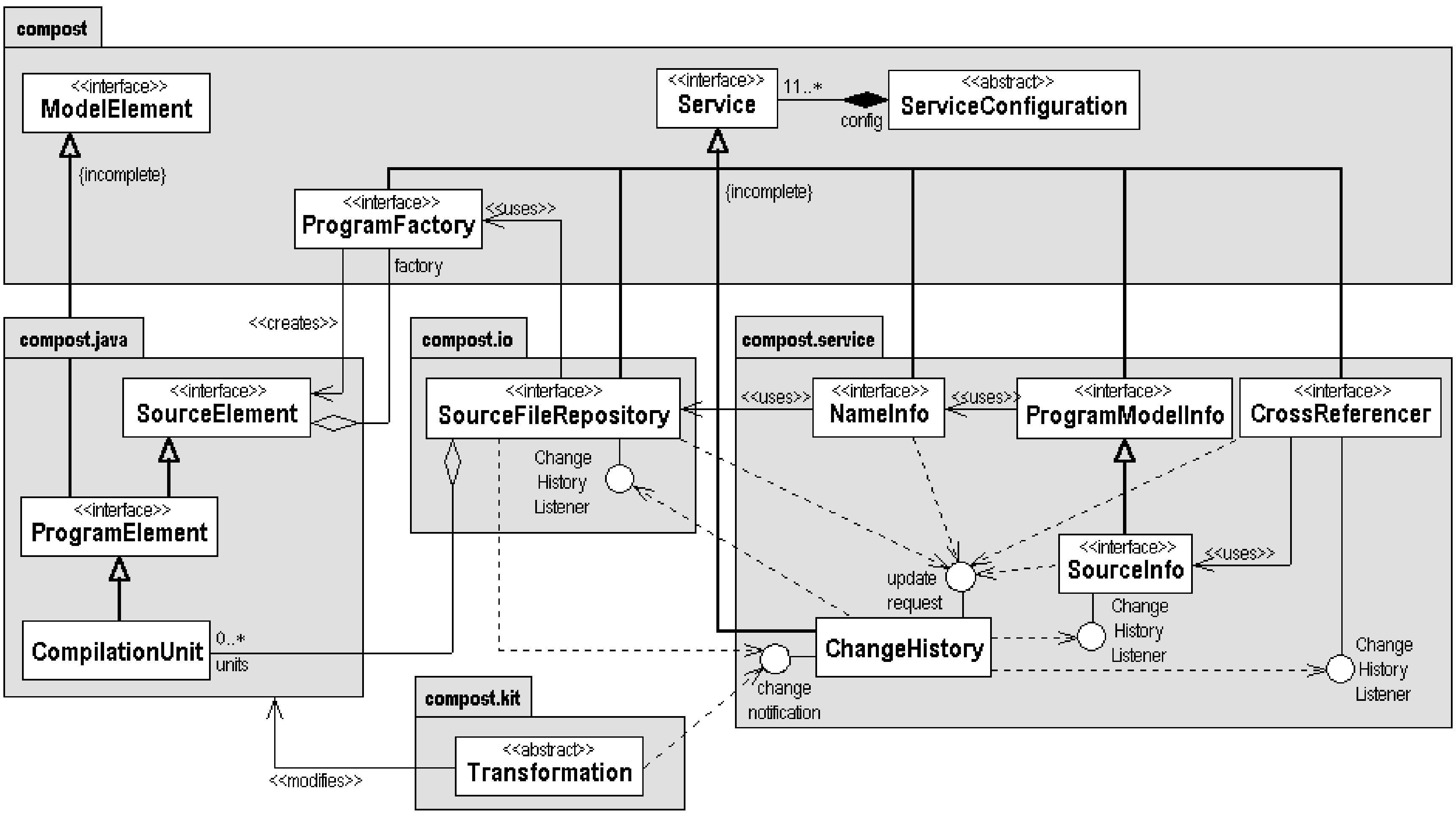
- Change notification optimization:
  - Delay changes in a queue to avoid traversals.
  - Tag subtree changes as minor to avoid traversals.
  - Clear queue after notification.
- Rollback support:
  - Keep changes on a stack.
  - To roll back, reverse changes and create reports for changes that already have been reported.
  - Clear stack after commit (or before overflow).



# Model Elements and Services/Subtools



# Dataflow between Subtools




Prof. Uwe Aßmann, Design Patterns and Frameworks

# Change Impact Analysis

Efficient updates of reference information:

- If something changes, what are possibly effected declarations and references?
  - Examples follow...
- Does the target of a reference really change?
  - Access the former result to compare: Cache everything!
  - Only verified cached results can be used for the update.
  - May lead to new change tests, but is guaranteed to stop.
- Update cached information efficiently.
  - Reference sets instead of lists.

# Examples for Change Impacts

- If an expression changes...
  - ...its parent reference might change.
- If a method declaration changes...
  - ...all inherited, inheriting, inner, outer, possibly overloaded and possibly overloading method references with compatible name and signature might change.
- If a subtype relation changes...
  - ... references might change as if all former and now inherited member declarations changed.

# Transformation Model

- Reify as objects (Command/Objectifier Pattern of GOF).
  - Transformations must be managed for nested transactions.
  - Transformations often have to access analysis results and generated code fragments of subtransformations.
- Each transformations can yield a problem report or assert program states (e.g. compileable, or idempotent)

# Transformation Composition

- Transformations may have dependencies.
- Ideal Case: 2-pass (analyze - transform)
  - Combinations result in another 2-pass operation.
  - This case is not too rare: Changes of disjoint declarations will affect disjoint references.
- Usual Case: 1-pass (analyze & transform)
  - Parent transformation must update local data.
  - Restart traversal at the “first” change location.
  - Check idempotency to ensure termination.
  - Worst case: Restart always - $O(n^2)$

# Extensibility: Program Models

- New Program Model Entities
  - Add entities as subclasses of the proper types (ModelElement if nothing else applies).
  - Optionally add a management service to locate or create the new entities or keep them persistent.
- Examples:
  - Design pattern instances documenting interesting structures for quick retrieval (change of design).
  - Box & Hook Model maintained by a BoxInfo.

# Extensibility: Metaprograms

- New Analyses
  - Add as auxiliary class/method if there is no need for cached data.
  - Create and register a service to participate at the change propagation, if you need incrementality.
- New Transformations
  - Simply add new subclasses of Transformation.
- Examples
  - Reachability analysis (conservative version is local)
  - Composers

# How to Refactor Everything? (1)

What kind of document can we transform?

- Strongly typed source code.
- Makefiles?
- XMI documents?
- HTML pages?
- A spreadsheet document?

They all obey certain formal rules...

# How to Refactor Everything? (2)

- The RECODER change mechanisms operate on syntactic level.
- Formal documents are structured.
  - Terminal nodes, non terminal nodes, containment relation forming a tree.
  - Syntax Trees, XML Documents.
- The architecture works for syntactic documents, if we add content type handlers.

# How to Refactor Everything? (3)

- Formal documents have a static semantic.
  - Different node types (e.g. Identifier, Operator)
  - Statically computable n-ary predicates
    - e.g. isAbstract(Method), refersTo(Reference, Definition)
  - Computation of these properties, relations etc. is highly specific.

```
class X {
  /*nonsense*/
  X myself;
}
```

```
<A NAME="X"></A>
nonsense
<A HREF="#X">myself</A>
```

# How to Refactor Everything? (4)

- Except for some parts of the parser, RECODER has been created manually.
- We need toolkits that create
  - a parser (including comment assignment and indentation information),
  - an unparser (customizable),
  - incremental semantic analyzers,
  - atomic type-safe transformations

  from some suitable definitions (AGs?)

# The End

- Talk courtesy to Andreas Ludwig (2004)
- Work on RECODER started 1997 (A. Ludwig)
  - Attempt to commercialize in 2001-2 (Sweden)
  - Open source since 2001
  - Still alive
- A. Ludwig. Automatische Anpassung von Software. Dissertation. Universität Karlsruhe, 2002.