Feature Modeling Based Multi-Paradigm Design for AspectJ

Valentino Vranić

`www.fiit.stuba.sk/˜vranic, vranic@fiit.stuba.sk`

Institute of Informatics and Software Engineering
Faculty of Informatics and Information Techologies
Slovak University of Technology

L3S Info-Lunch Presentation — October 1, 2004

Introduction

- Software development process: an application (problem) to solution domain mapping
- Software development paradigm: how to express application domain concepts in terms of solution domain concepts
- Solution domain concepts correspond to programming language mechanisms
- Choosing the appropriate paradigm is an important issue
- Individual solution domain concepts (e.g., a class in Java) may be regarded as paradigms

Presentation Overview

- The concept of paradigm
- Feature modeling
- Multi-paradigm design with feature modeling ($\mathrm{MPD}_{\mathrm{FM}}$)
- Paradigm modeling in $\mathrm{MPD}_{\mathrm{FM}}$
- Transformational analysis in $\mathrm{MPD}_{\mathrm{FM}}$
- $\mathrm{MPD}_{\mathrm{FM}}$ evaluation
- Aspect-oriented modeling and $\mathrm{MPD}_{\mathrm{FM}}$
- Summary and further work

The Concept of Paradigm

- The original meaning: example or pattern
- Scientific paradigm[a]
- Paradigms of programming and software development[b]
  - The essence of a software development process
  - A “popular meaning of the word”: large-scale paradigms[c]
  - Procedural, logic, functional, object-oriented paradigm. . .

---

[a] T. S. Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press, Chicago, 1970.

[b] R. W. Floyd. The paradigms of programming. *Communications of the ACM*, 22(8), 1979.

[c] J. O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1999.

Small-Scale Paradigms

- Programming language perspective
- Configurations of commonality and variability
- Scope, commonality, variability, and relationship (SCVR) analysis[a]
- An example: the procedure paradigm

Scope: a collection of similar code fragments, each to be replaced by a call to some new procedure

Commonality: the code common to all fragments

Variability: the “uncommon” code; variabilities are handled by procedure parameters or custom code

---

[a] J. O. Coplien et al. Commonality and variability in software engineering. *IEEE Software*, 15(6), Nov. 1998.

Multi-Paradigm Software Development

- Two issues:
  - Making multiple paradigms available: multi-paradigm languages (e.g., Leda[a])
  - Choosing an appropriate paradigm for the problem being solved: multi-paradigm design
- Multi-paradigm design methods
  - Multi-paradigm design method for Leda[b]
  - Multi-paradigm design (for C++)[c]

---

[a] T. A. Budd. *Multiparadigm Programming in Leda*. Addison-Wesley, 1995.

[b] C. D. Knutson et al. Multiparadigm design of a simple relational database. *ACM SIGPLAN Notices*, 35(12), Dec. 2000.

[c] J. O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1999.

Multi-Paradigm Design (MPD)

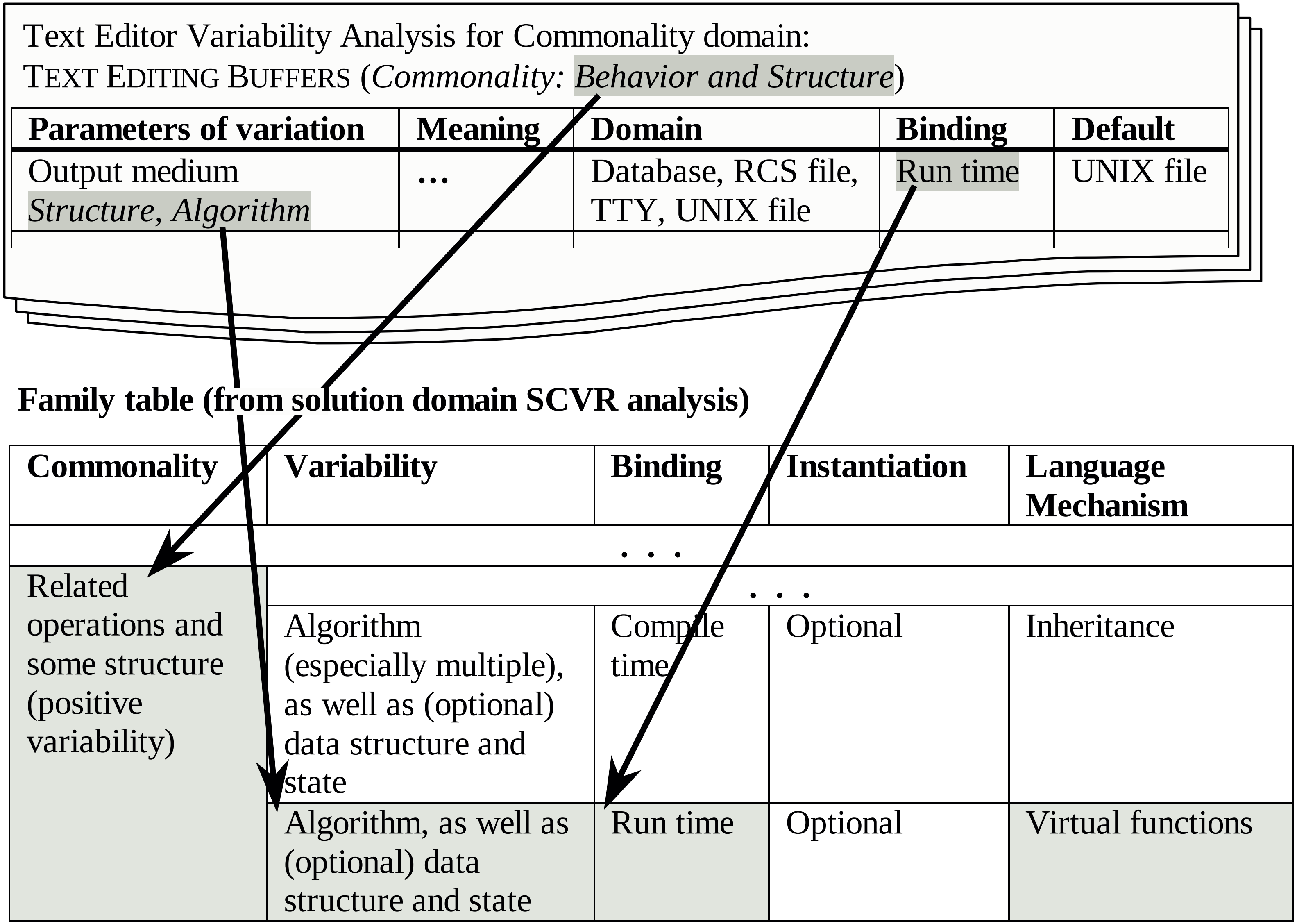
- MPD (for C++)[a] treats the solution domain in the same manner as the application domain (SCVR analysis)
- Both application and solution domain models are expressed mainly by tables
- Transformational analysis is preformed as a mapping between the tables
- Code design yields a code skeleton

---

[a] J. O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1999.

Transformational Analysis in MPD

Variability tables (from application domain SCVR analysis)



Text Editor Variability Analysis for Commonality domain:
TEXT EDITING BUFFERS (*Commonality: Behavior and Structure*)

Parameters of variation	**Meaning**	**Domain**	**Binding**	**Default**
Output medium *Structure, Algorithm*	…	Database, RCS file, TTY, UNIX file	Run time	UNIX file

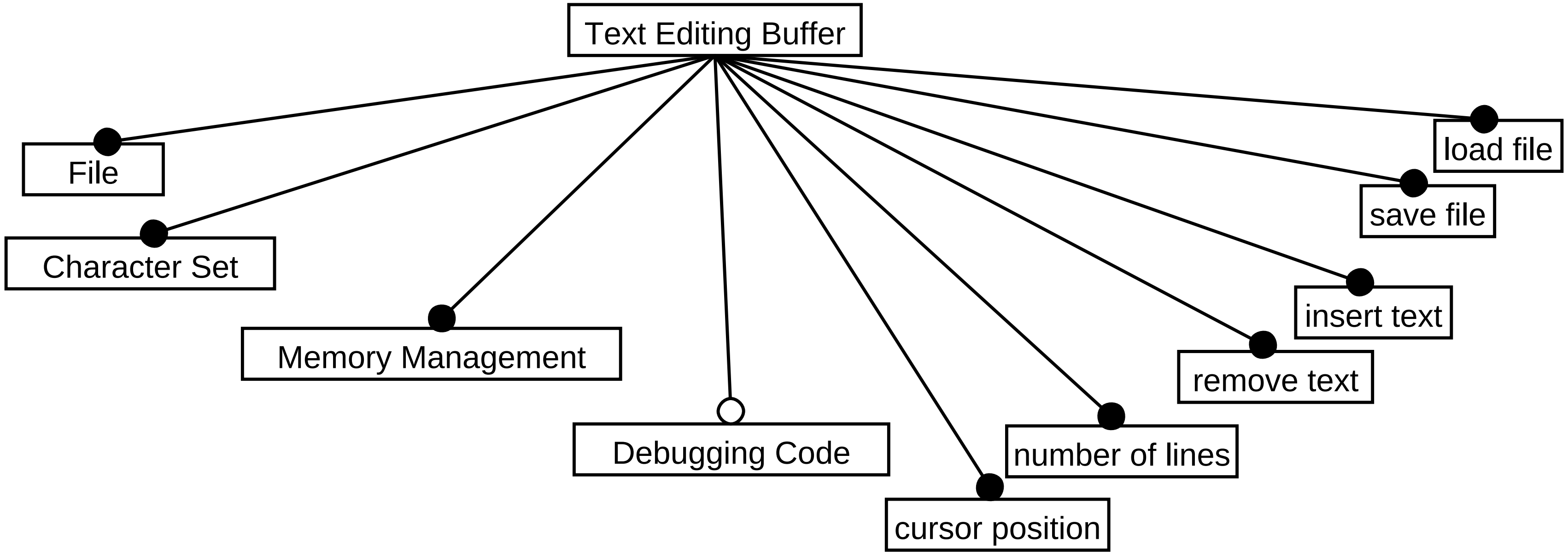
Family table (from solution domain SCVR analysis)

Commonality	**Variability**	**Binding**	**Instantiation**	**Language Mechanism**
. . .				
Related operations and some structure (positive variability)	. . .			
	Algorithm (especially multiple), as well as (optional) data structure and state	Compile time	Optional	Inheritance
	Algorithm, as well as (optional) data structure and state	Run time	Optional	Virtual functions

Feature Modeling

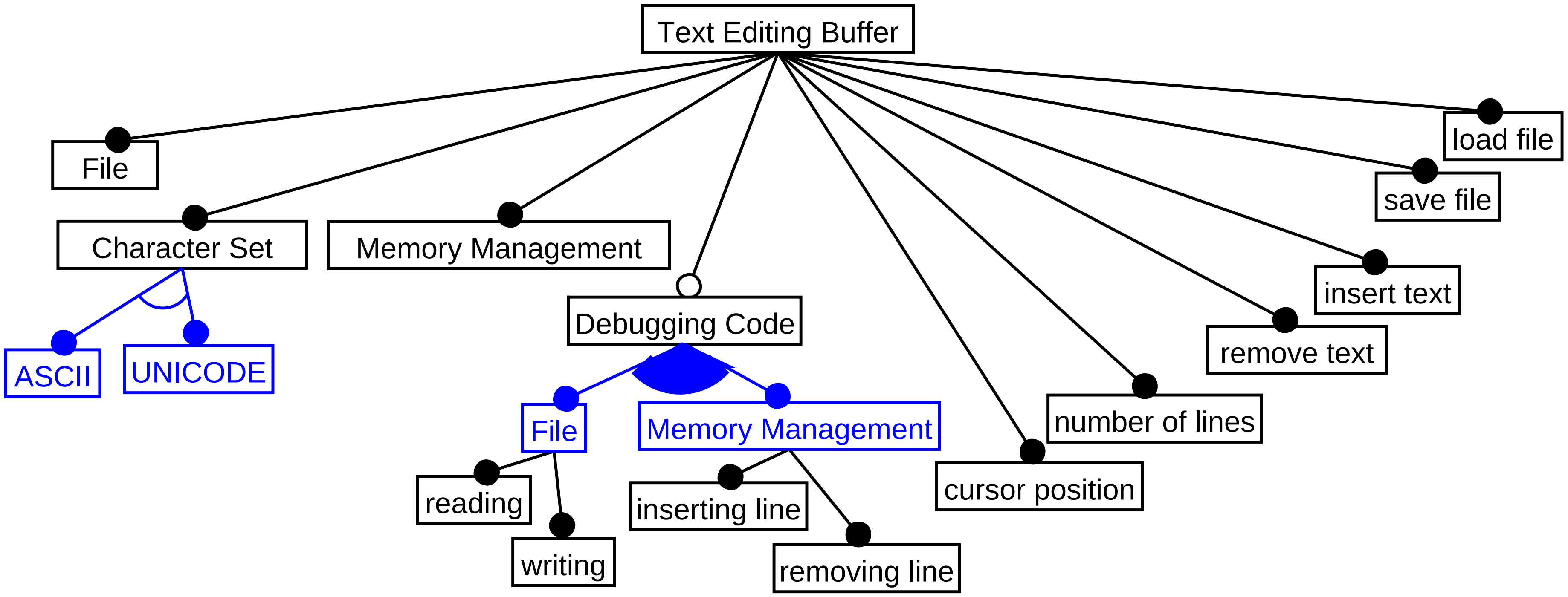
- Captures feature interdependencies and variability
- Feature model: a set of feature diagrams plus further information
- Based on the notions of domain, concept, and feature
  - Features: common and variable
  - Concept instances: concept specializations
- Different notations being used, such as FODA, ODM, Czarnecki-Eisenecker, and feature modeling for multi-paradigm design

Feature Variability



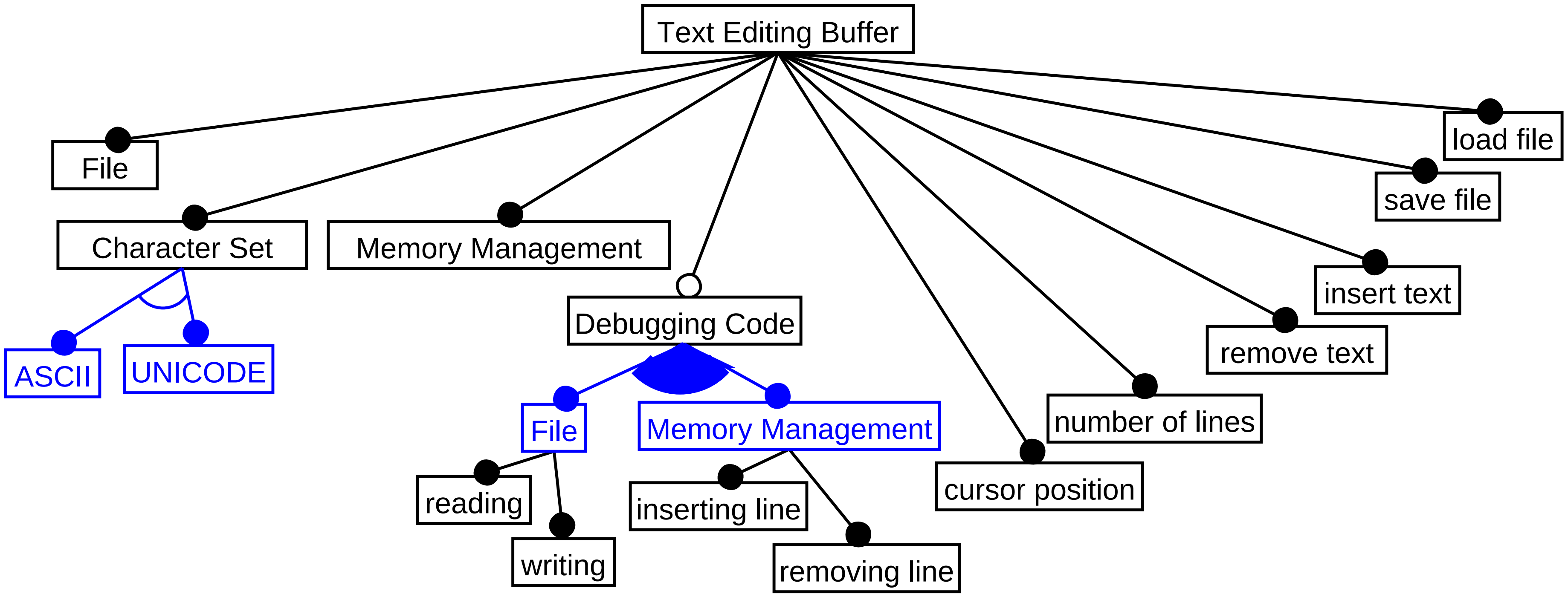
- Mandatory features (filled circle ended edges)
- Optional features (empty circle ended edges)

Feature Variability



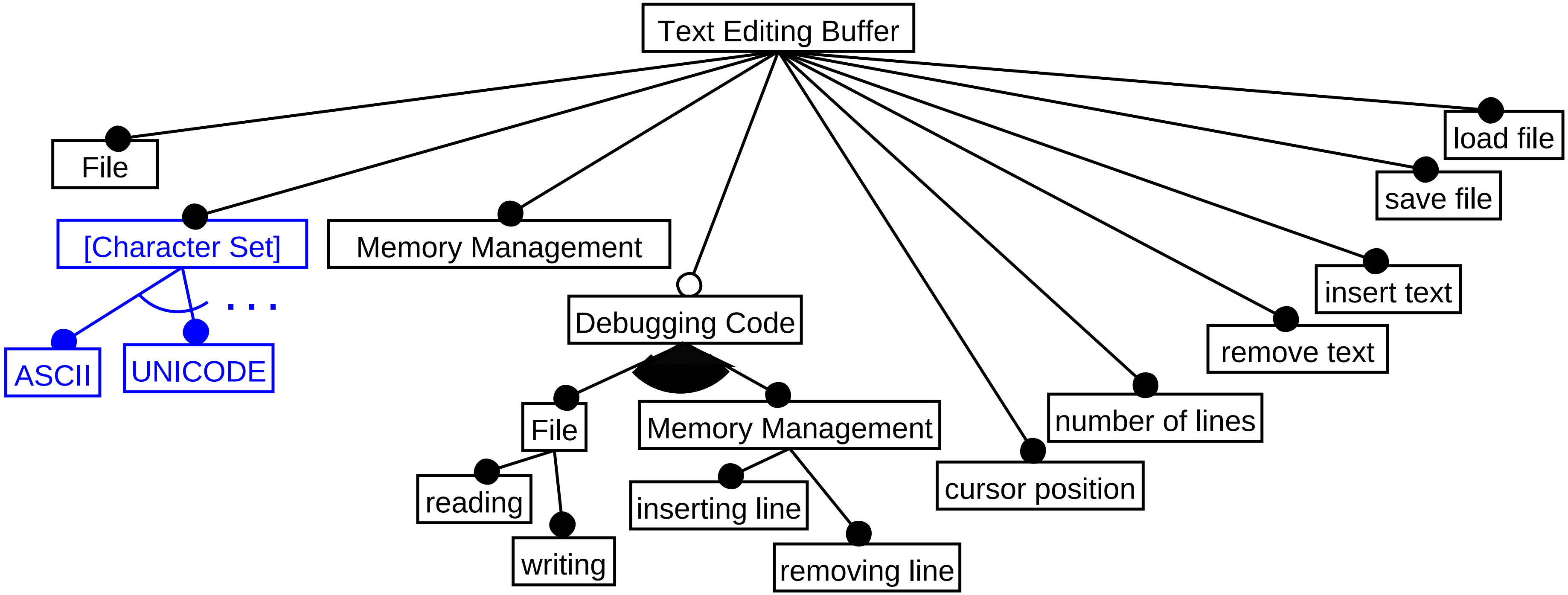
- Alternative features (empty arc)
- Or-features (filled arc)

Feature Variability



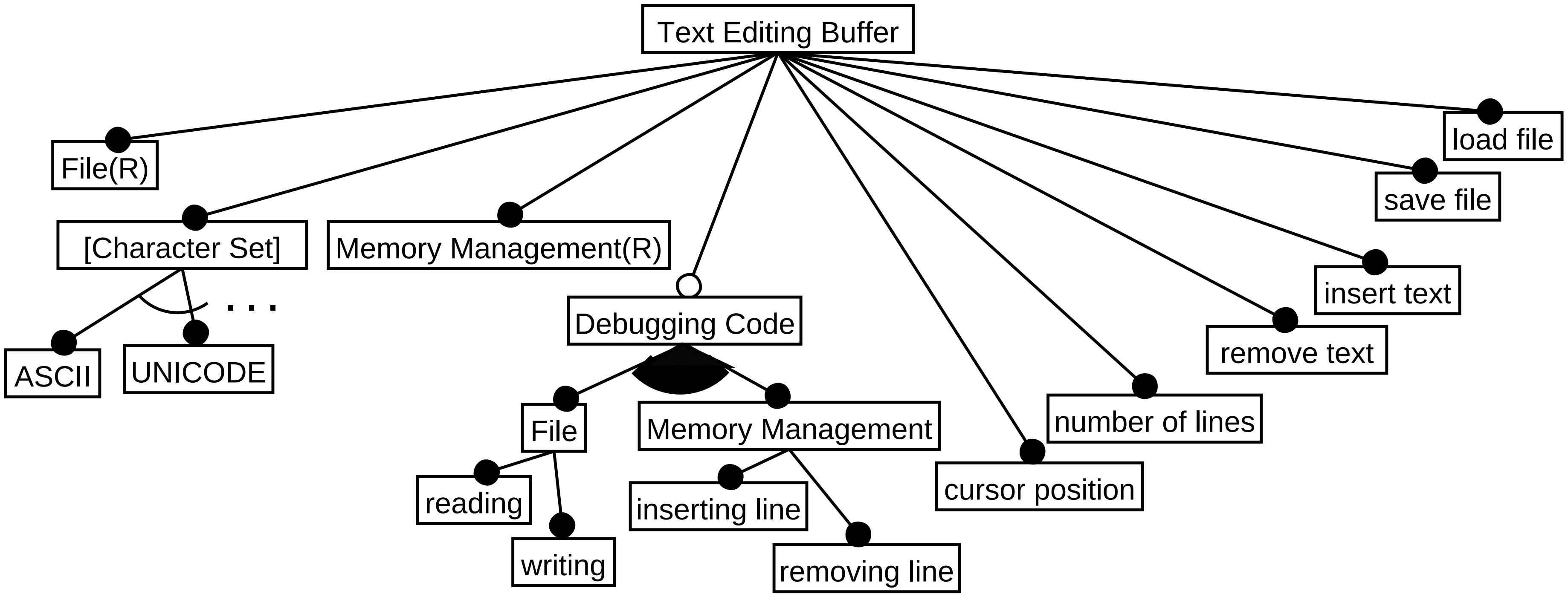
- Edges combine with arcs
  - Mandatory alternative / optional alternative features
  - Mandatory or- / optional or-features

Feature Variability



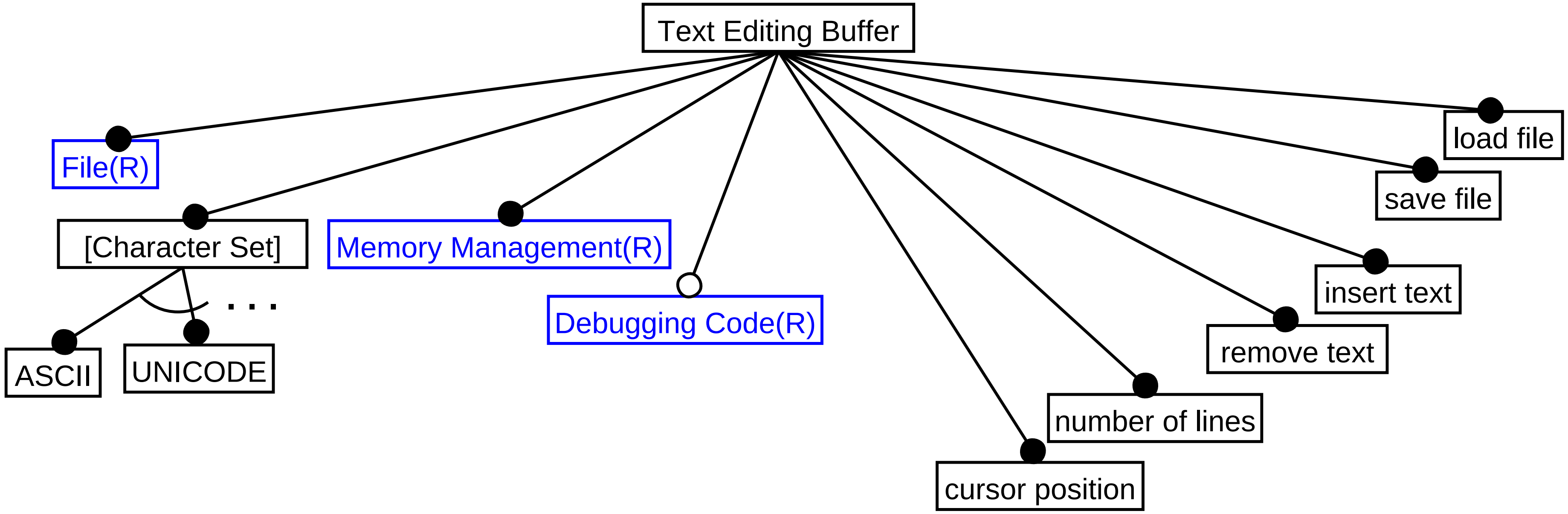
- Open features
  - Further variable subfeatures expected
  - Denoted by square brackets and, optionally, ellipsis

Feature Variability



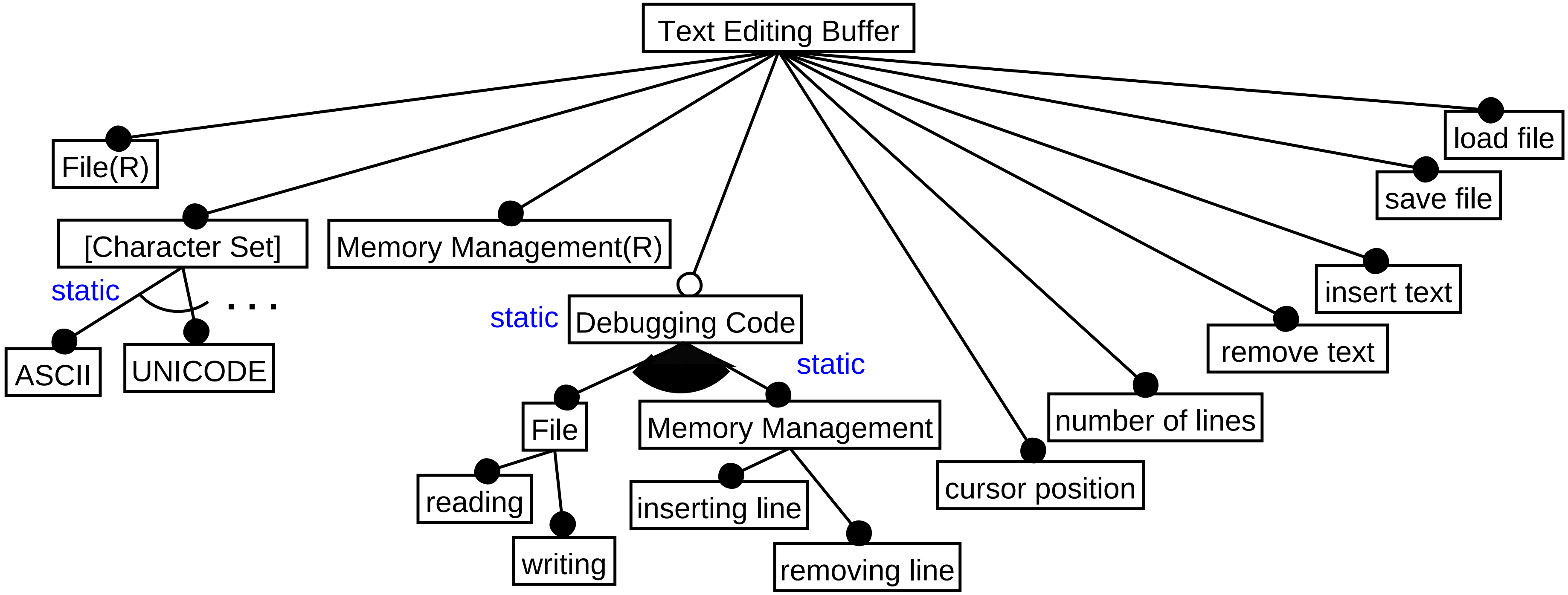
- Inclusion of a feature in a concept instance is stipulated by the inclusion of its parent
- Features of any variability type can appear at any level

Concept References



- Denoted by ® (appears as (R) in diagrams)
- Can be expanded as needed

Binding Time/Mode

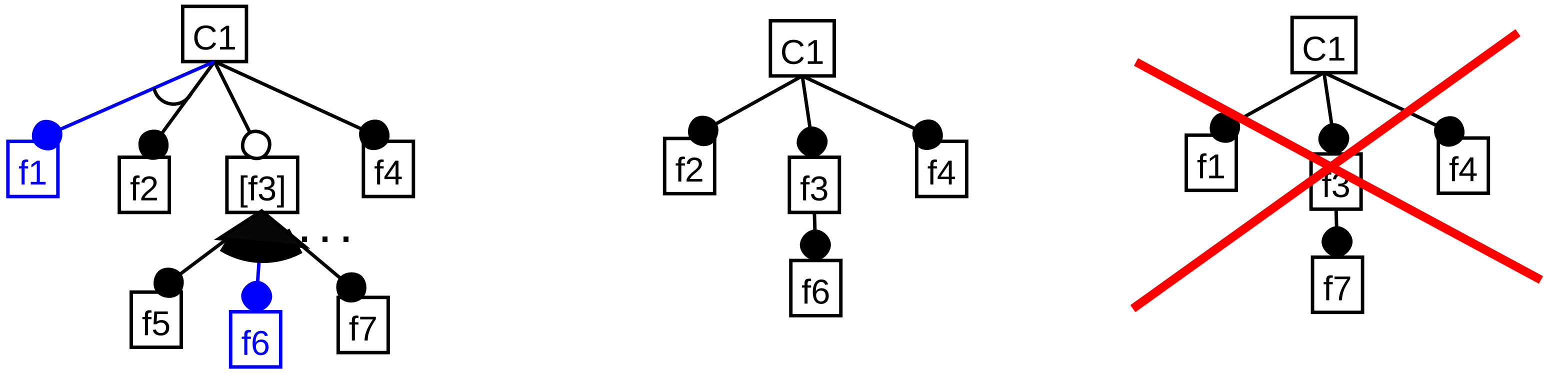


- When/how a feature is to be bound
- Usual binding times: source, compile, link, load, and run time
- Binding mode: static or dynamic

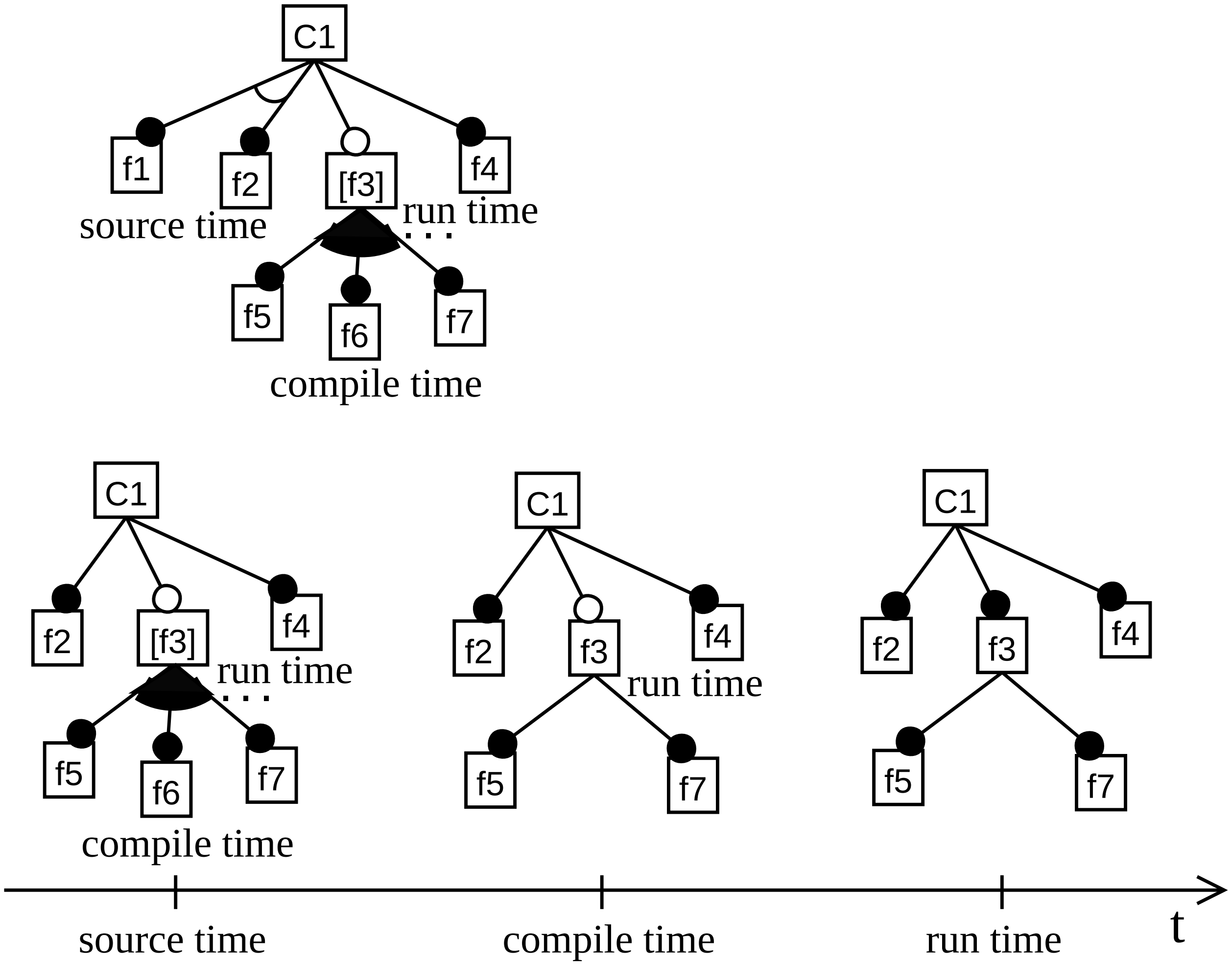
Further Information in Feature Models

- Information associated with concepts and features
  - Textual information: description, presence rationale, inclusion rationale, note
  - Binding time/mode
- Constraints and default dependency rules
  - A constraint example

$f1 \Rightarrow f6$



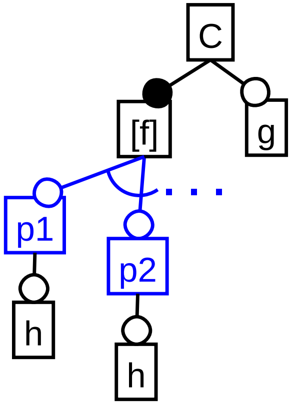
Concept Instantiation



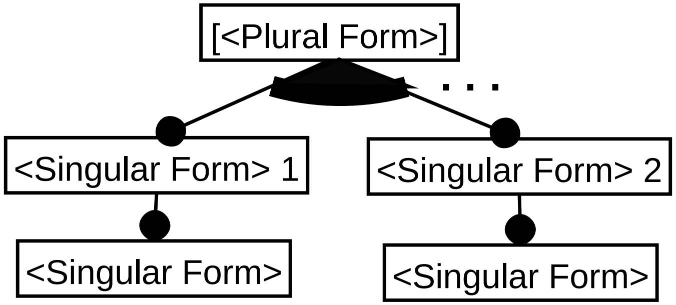
Parameterization in Feature Models

- Parameterized feature and concept names

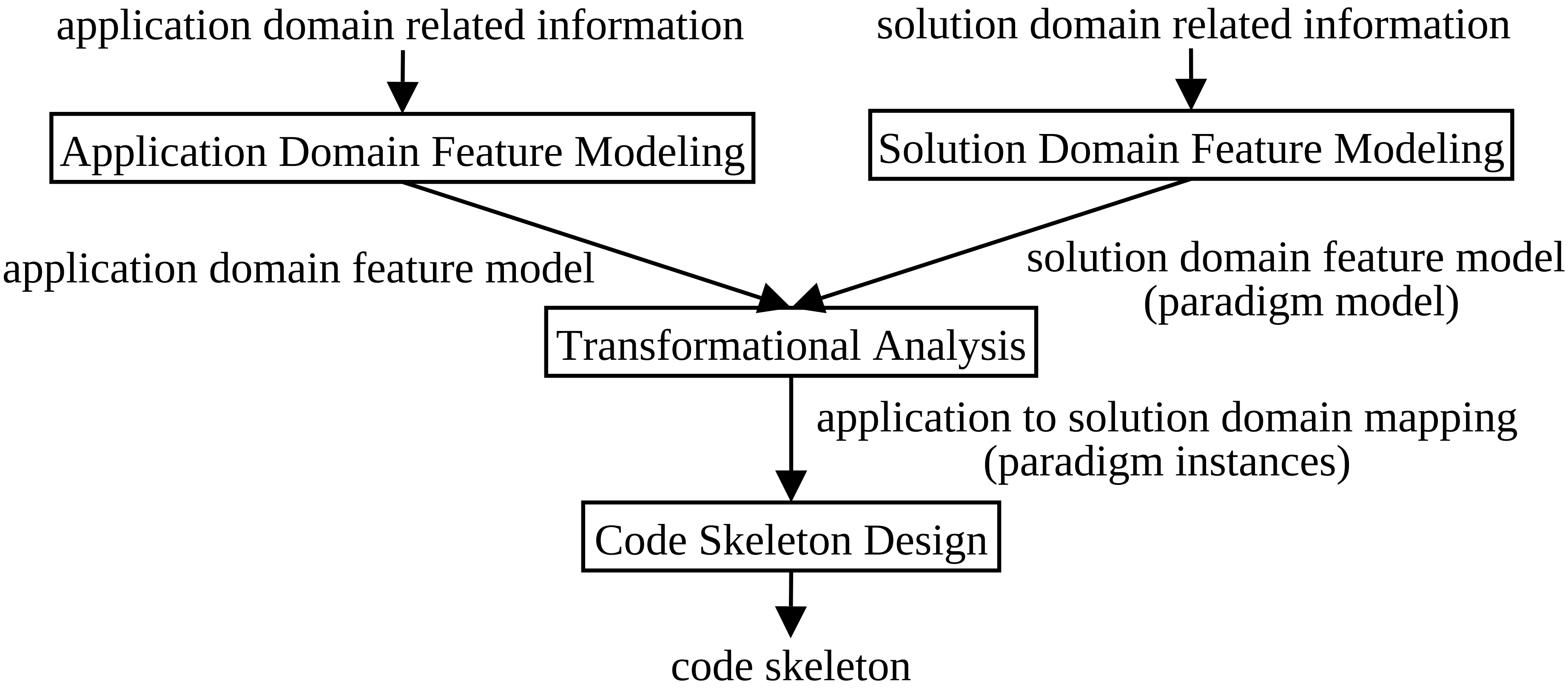
  Constraint: $\forall < i > \in N \; p < i > .h \; \underline{\vee} \; g$



- Parameterized concepts



MPD$_{FM}$ Activities



application domain related information

solution domain related information

Application Domain Feature Modeling

Solution Domain Feature Modeling

application domain feature model

solution domain feature model
(paradigm model)

Transformational Analysis

application to solution domain mapping
(paradigm instances)

Code Skeleton Design

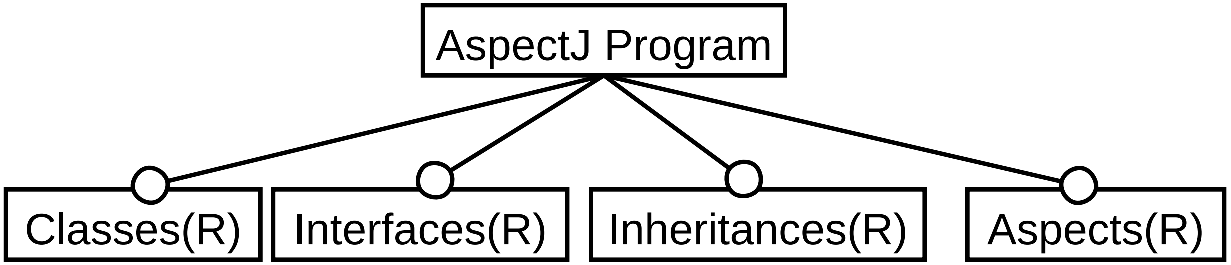
code skeleton

Paradigm Modeling in MPD_{FM}

- Identification of paradigms
  - Directly and indirectly usable paradigms
  - Hierarchy of paradigms
- Identification of binding times
  - A sequence of binding times provided by the solution domain
  - Usual binding times: source, compile, link, load, and run time
  - An AspectJ example: the method body—run time binding
- First-level paradigm model
- Modeling individual paradigms

First-Level Paradigm Model

- The solution concept
- Consists of directly usable paradigms
  - Subconcepts of the solution concept
  - Introduced as concept references (usually in plural)
  - Their variability and binding time should be determined
- An example: AspectJ first-level paradigm model



AspectJ Program

Classes(R)

Interfaces(R)

Inheritances(R)

Aspects(R)

Modeling Individual Paradigms

- Each paradigm is introduced in a separate feature diagram
  - Solution domain concepts
  - May reference each other
- Auxiliary concepts
  - Concepts referenced by paradigms
  - But not considered to be paradigms themselves
- Binding time (variable features)
- Instantiation is modeled by features

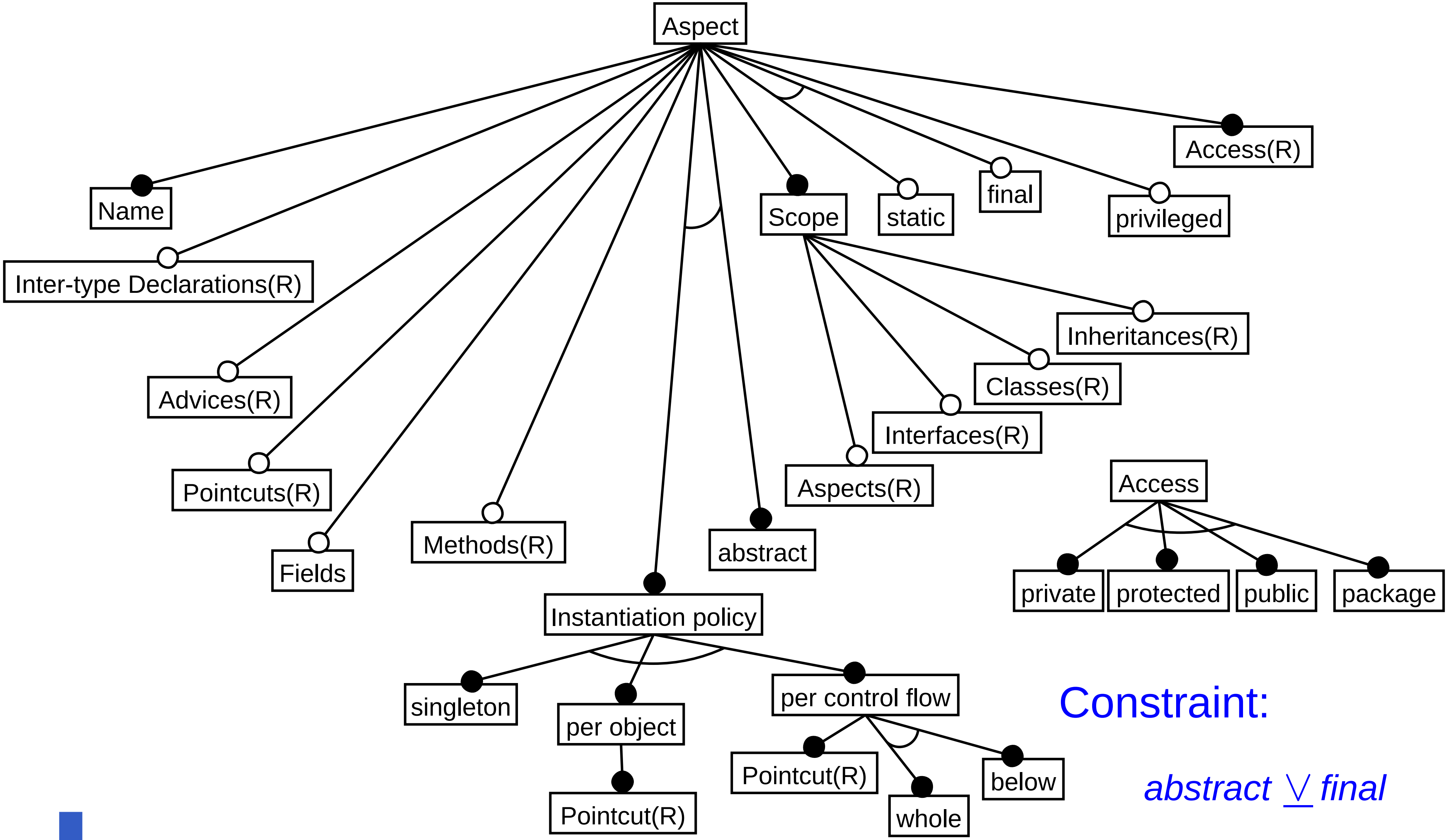
Structures and Relationships

- Structural paradigms correspond to main constructs (structures) of the programing language
- Relationship paradigms are about the relationships between programing language structures
- An application domain concept node in transformational analysis
  - Can match with the root of a structural paradigm
  - Cannot match with the root of a relationship paradigm

AspectJ Aspect-Oriented Paradigms

- Aspect-oriented programming
  - Modularization of crosscutting concerns
  - Useful for debugging, tracing, and synchronization in general
  - Application-specific aspects
- The aspect paradigm:
  - A structural paradigm (modularization)
  - A container of advices, pointcuts, and inter-type declarations; relationship paradigms (crosscutting concerns)

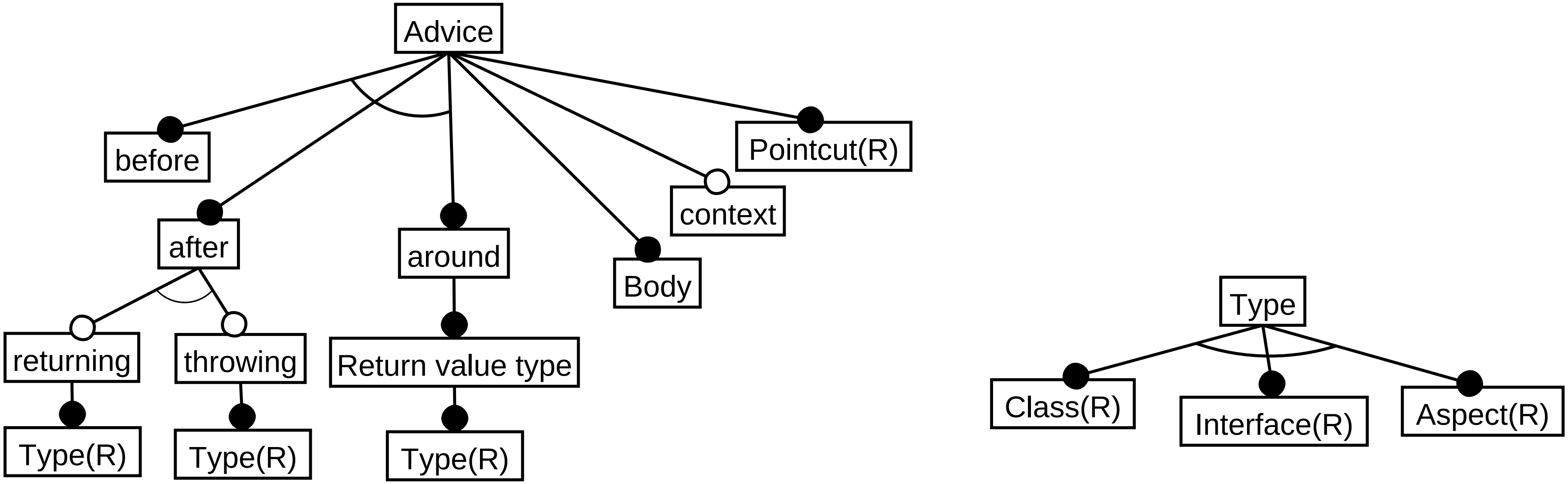
Aspect

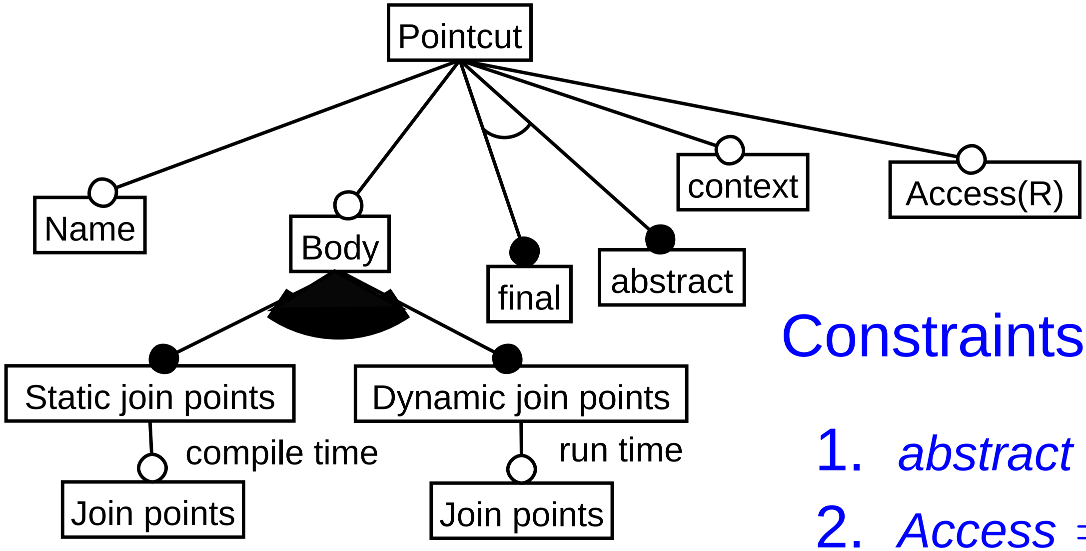


Constraint:

abstract $\veebar$ *final*

Advice and Pointcut





Constraints:

1. *abstract* $\underline{\vee}$ *Body*
2. *Access* $\Rightarrow$ *Name*

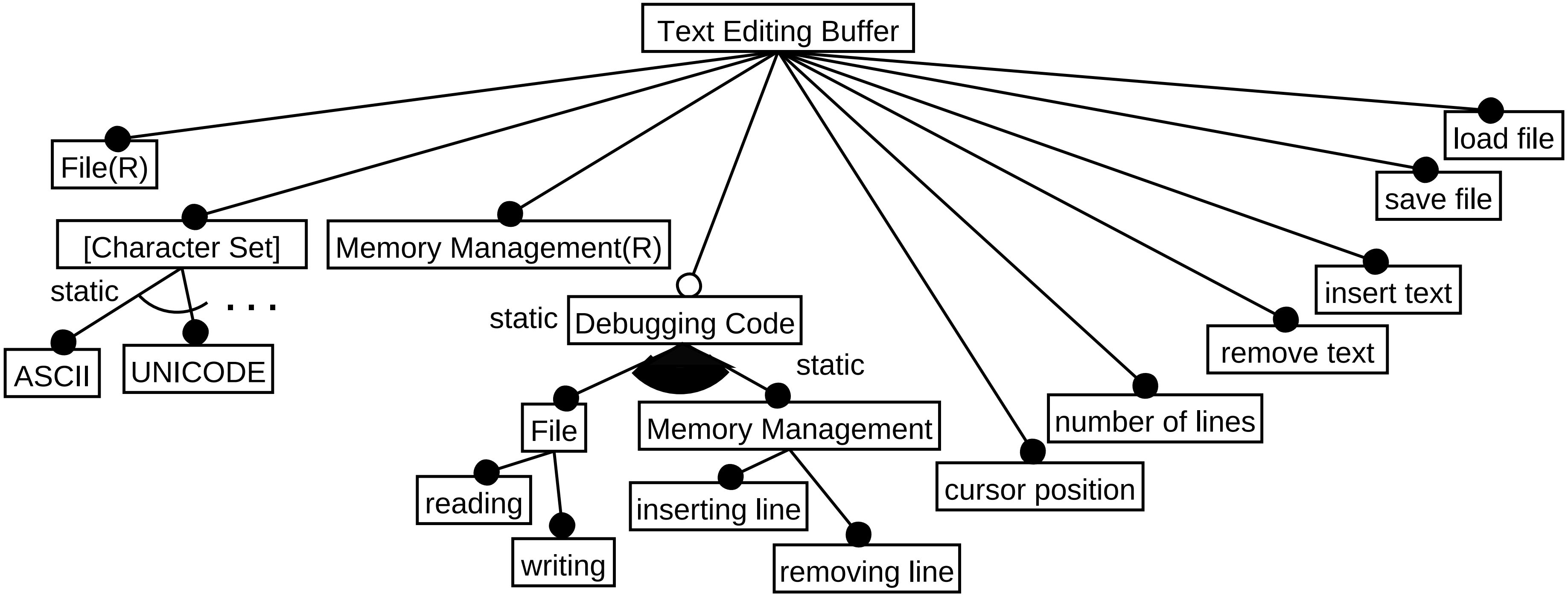
Transformational Analysis in MPD$_{FM}$

- Based on paradigm instantiation over application domain concepts at source time
- One application domain concept considered at a time
  1. Determine the structural paradigm of the application domain concept
  2. Determine the corresponding relationship paradigm for each unmapped relationship in it
- A creative process

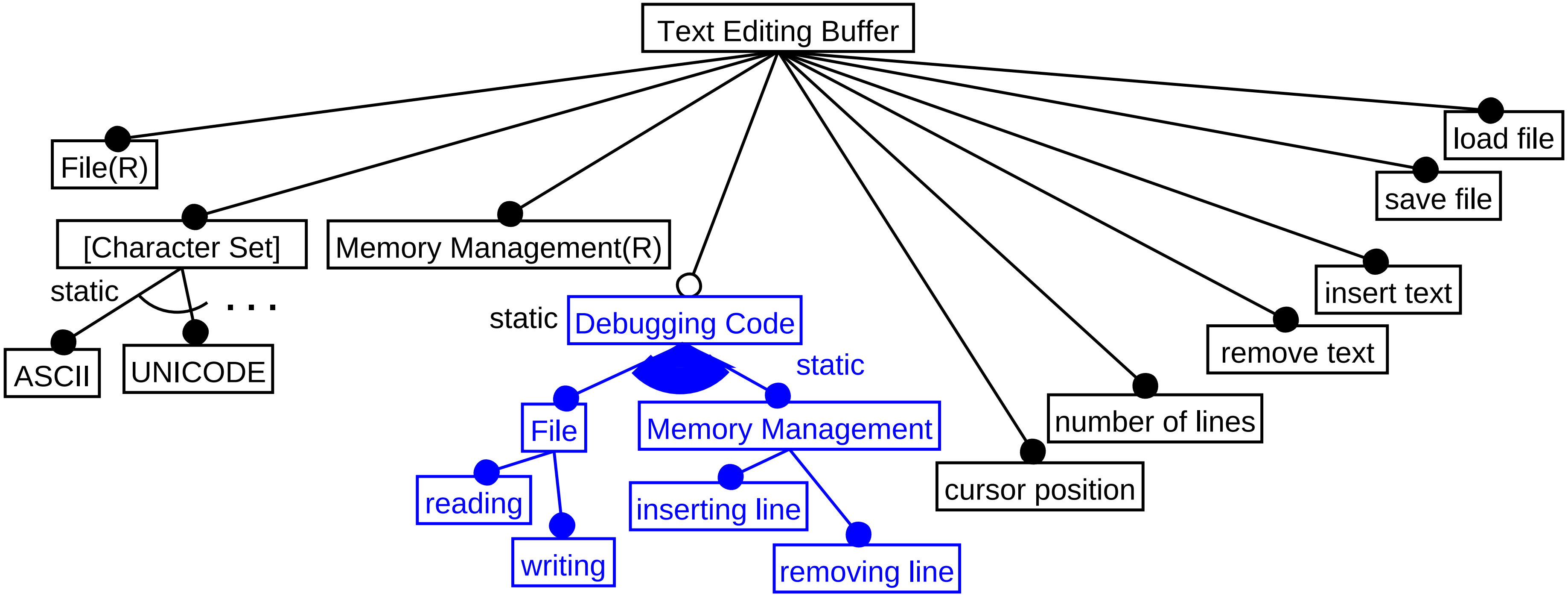
Paradigm Instantiation in MPD$_{FM}$

- Concept instantiation in MPD$_{FM}$
  - Viewed as concept specialization
  - Concept instances represented by feature diagrams
  - Takes into account binding time
- A bottom-up instantiation
- Inclusion of paradigm nodes stipulated by the mapping of the application domain concept nodes
  - Conceptual correspondence
  - Binding correspondence

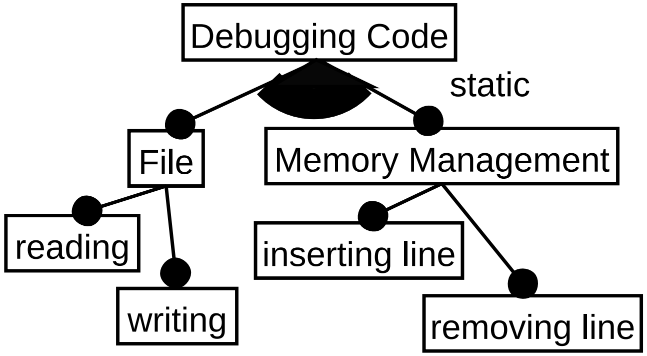
Transformational Analysis Example



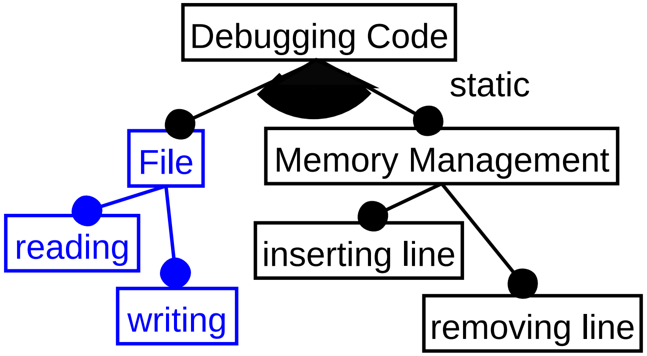
Transformational Analysis Example



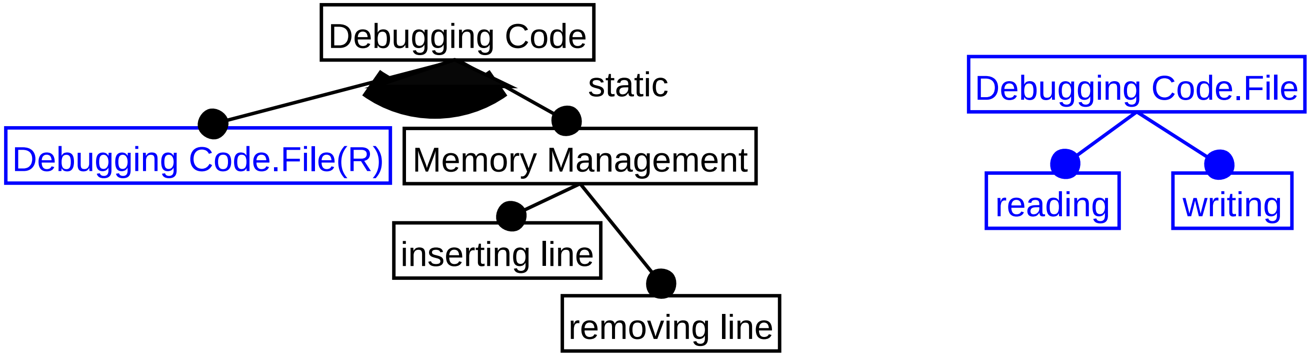
Transformational Analysis Example



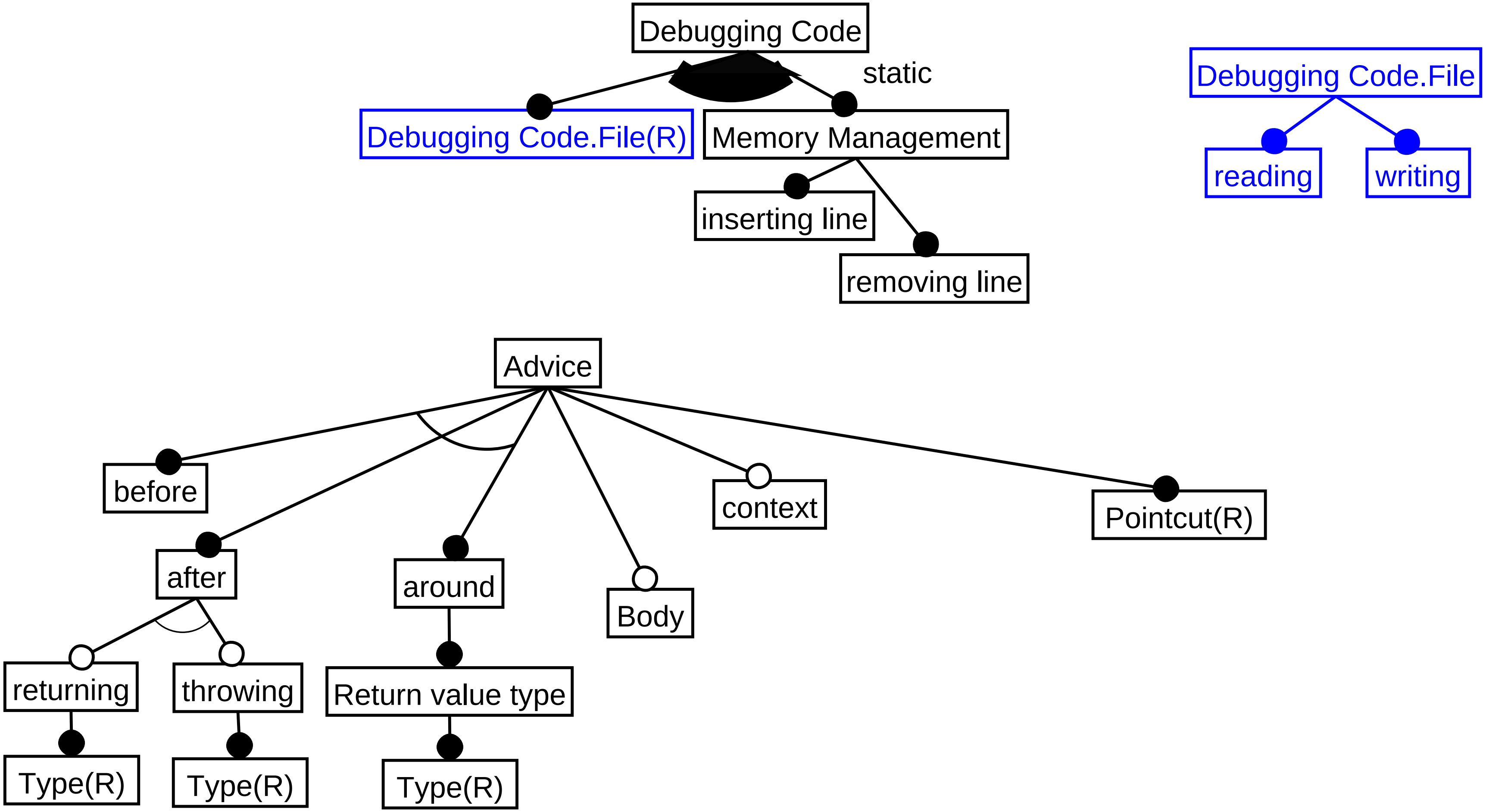
Transformational Analysis Example



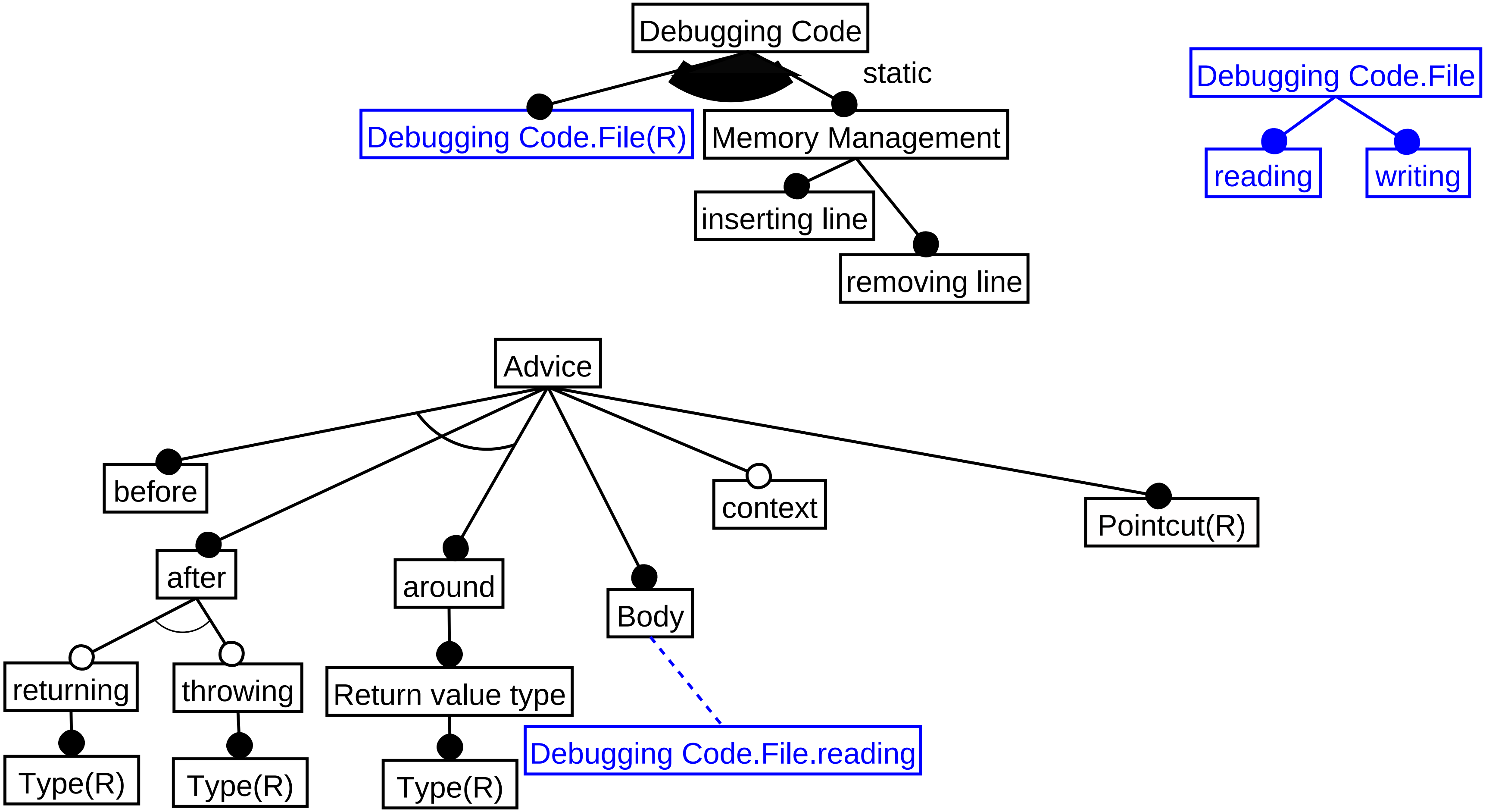
Transformational Analysis Example



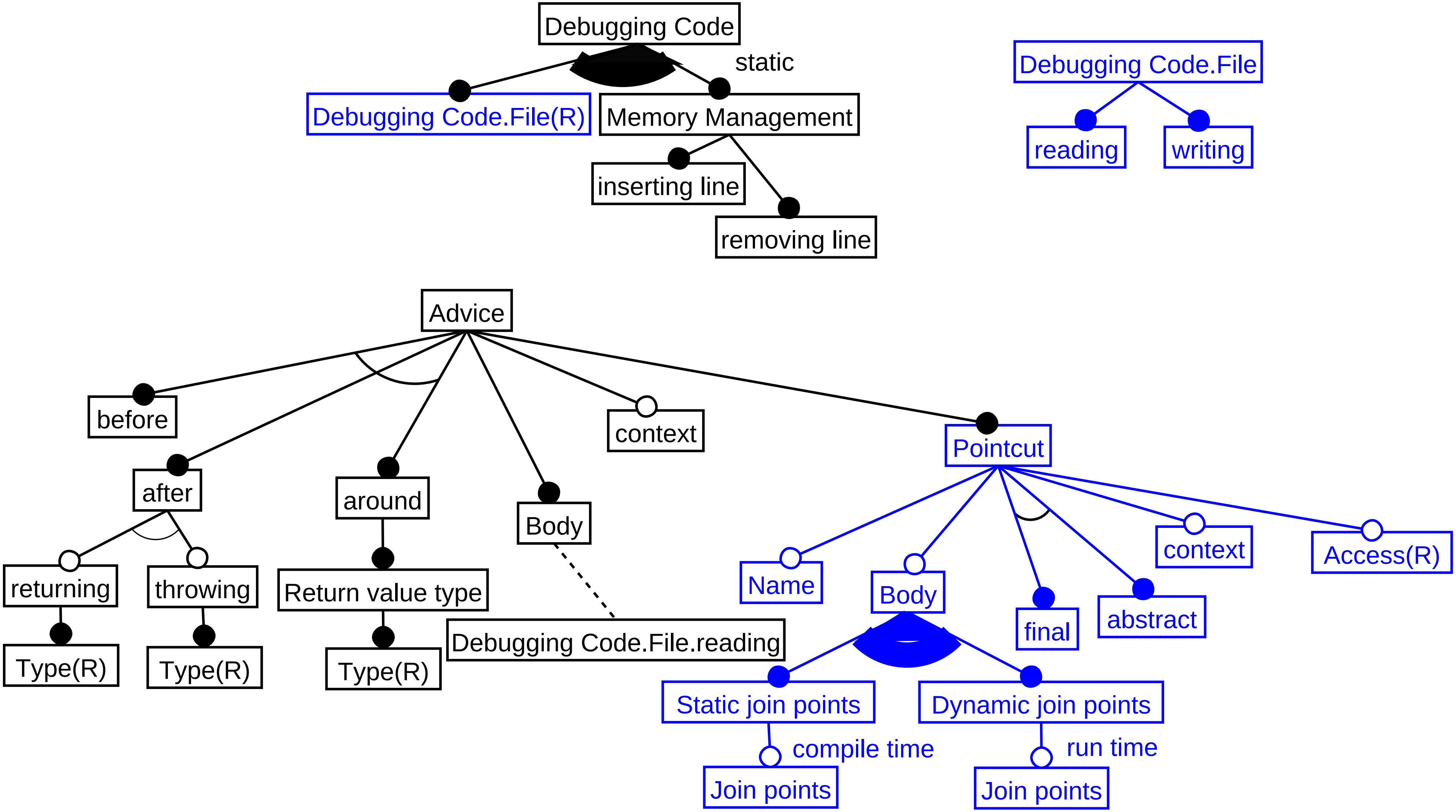
Transformational Analysis Example



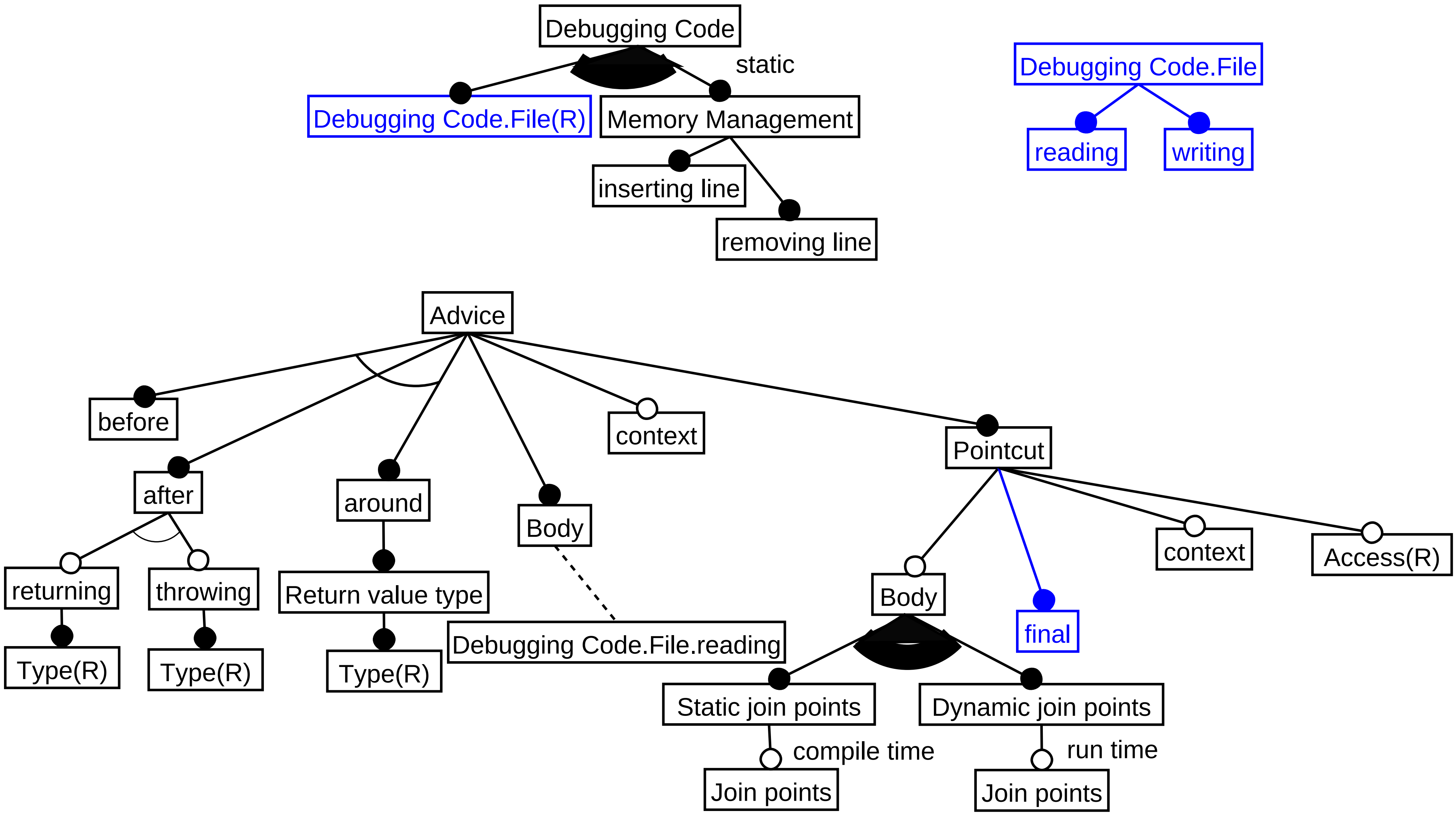
Transformational Analysis Example



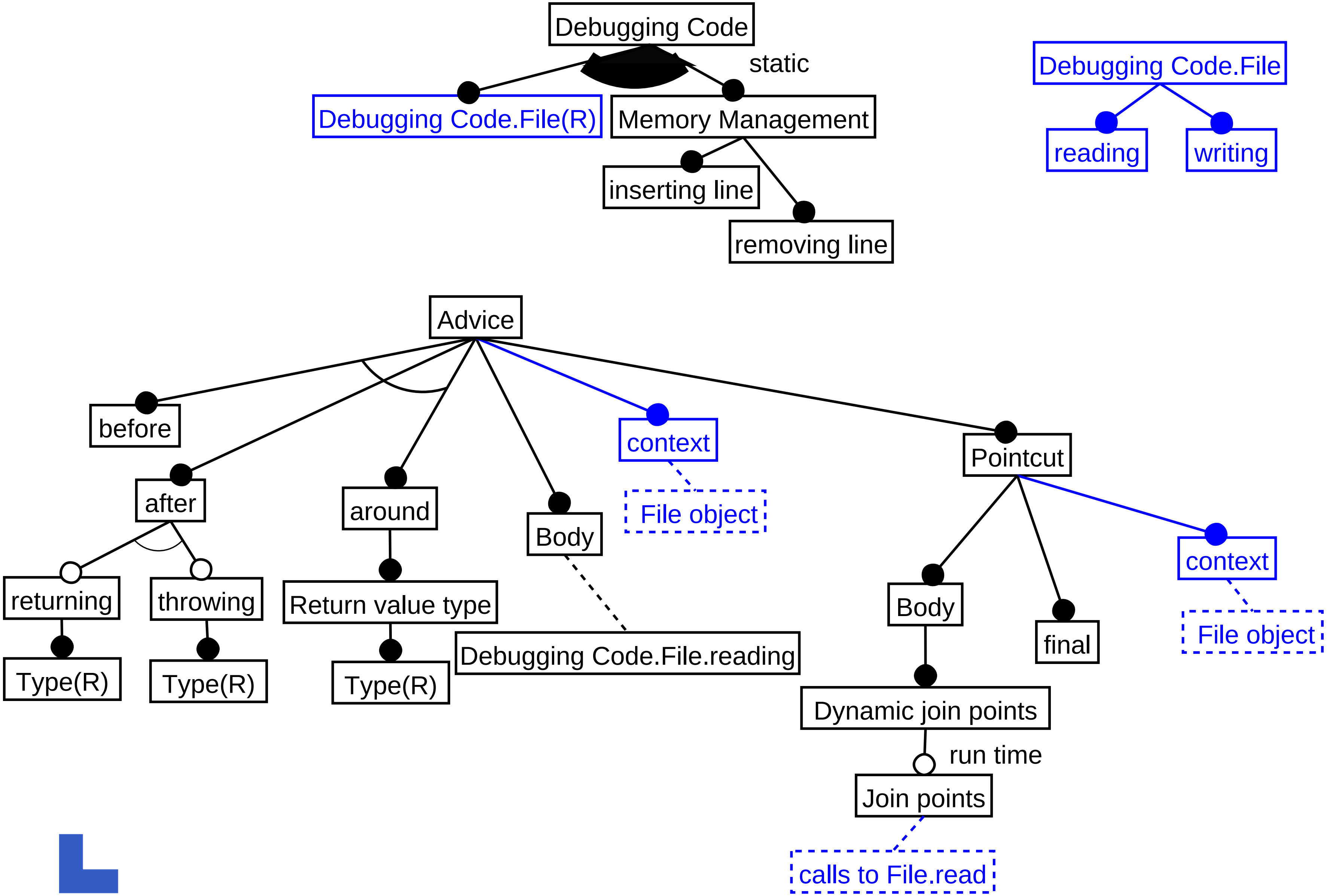
Transformational Analysis Example



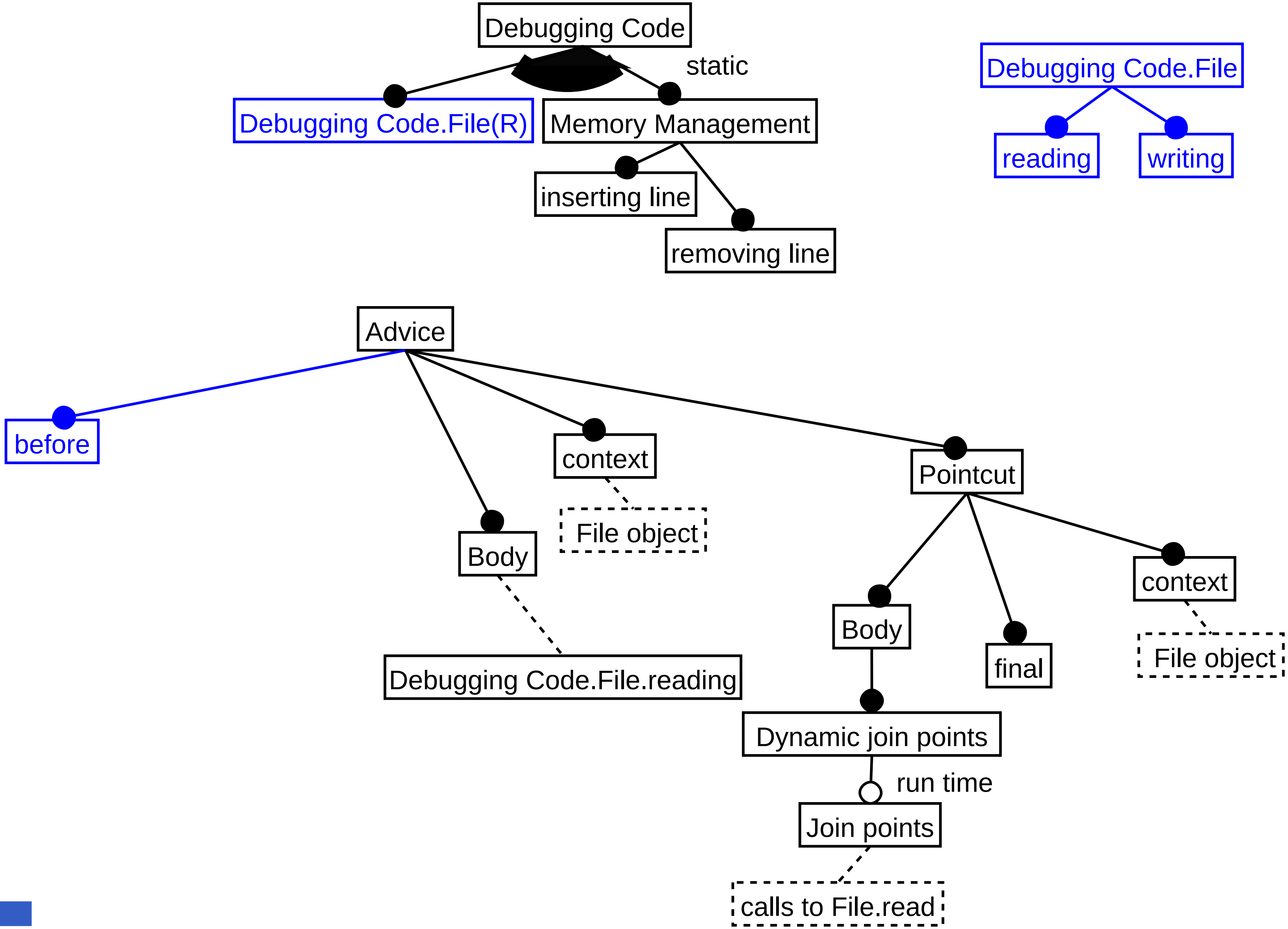
Transformational Analysis Example



Transformational Analysis Example



Transformational Analysis Example



Debugging Code

static

Debugging Code.File(R)

Memory Management

inserting line

removing line

Debugging Code.File

reading

writing

Advice

before

context

File object

Body

Debugging Code.File.reading

Pointcut

Body

final

context

File object

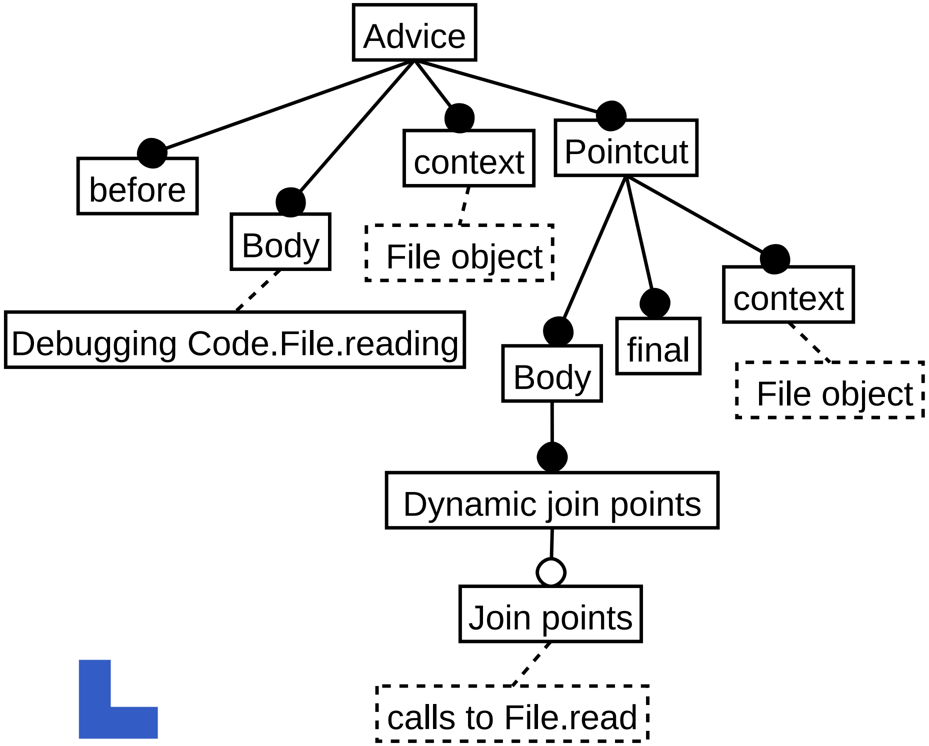
Dynamic join points

run time

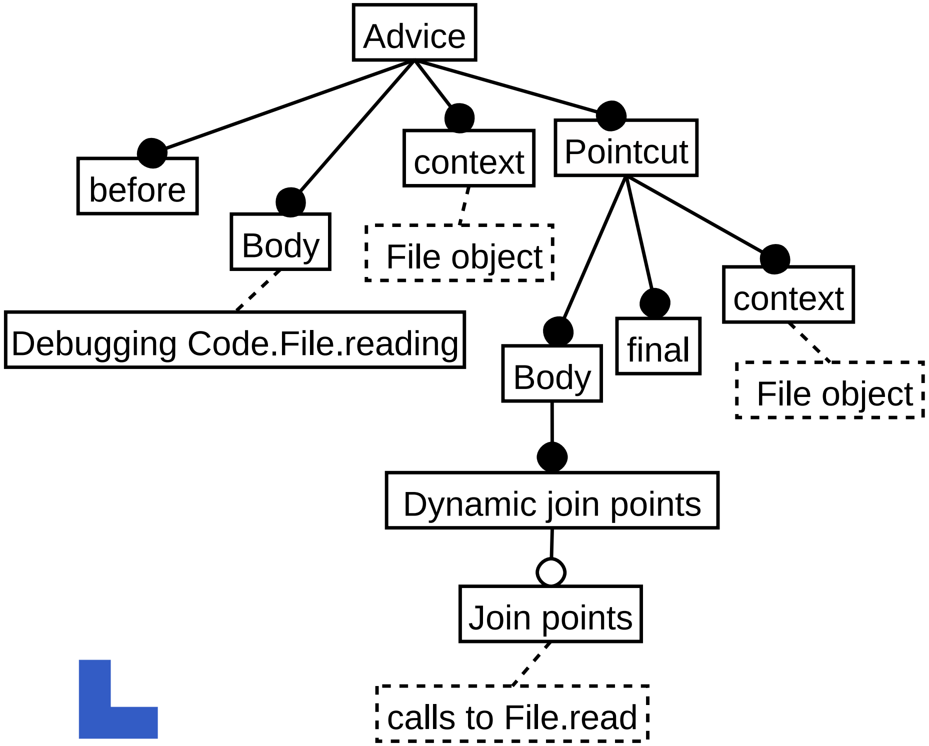
Join points

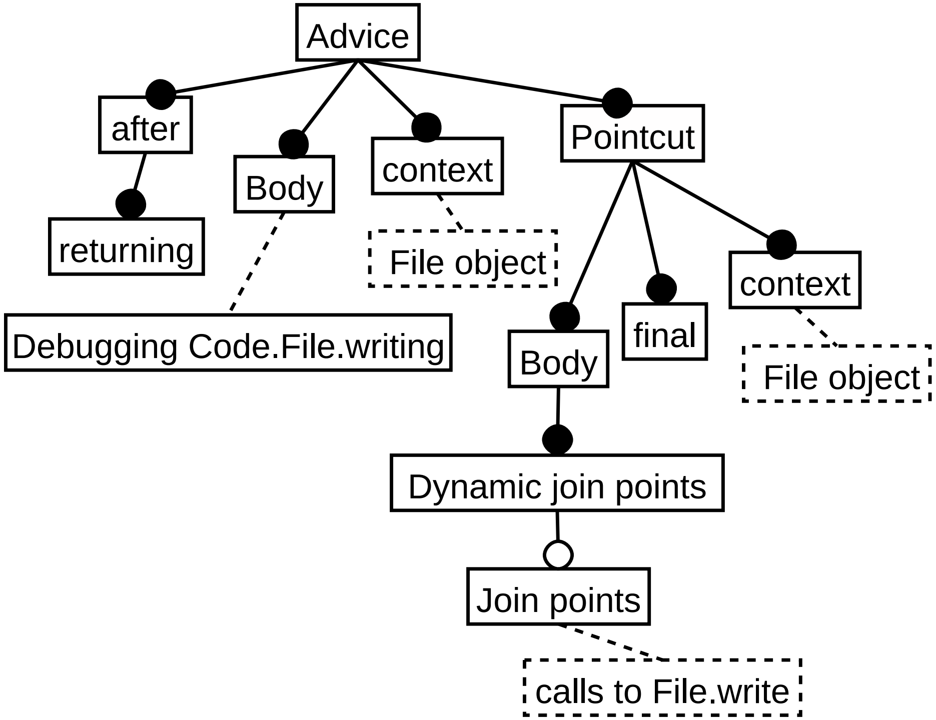
calls to File.read

Transformational Analysis Example

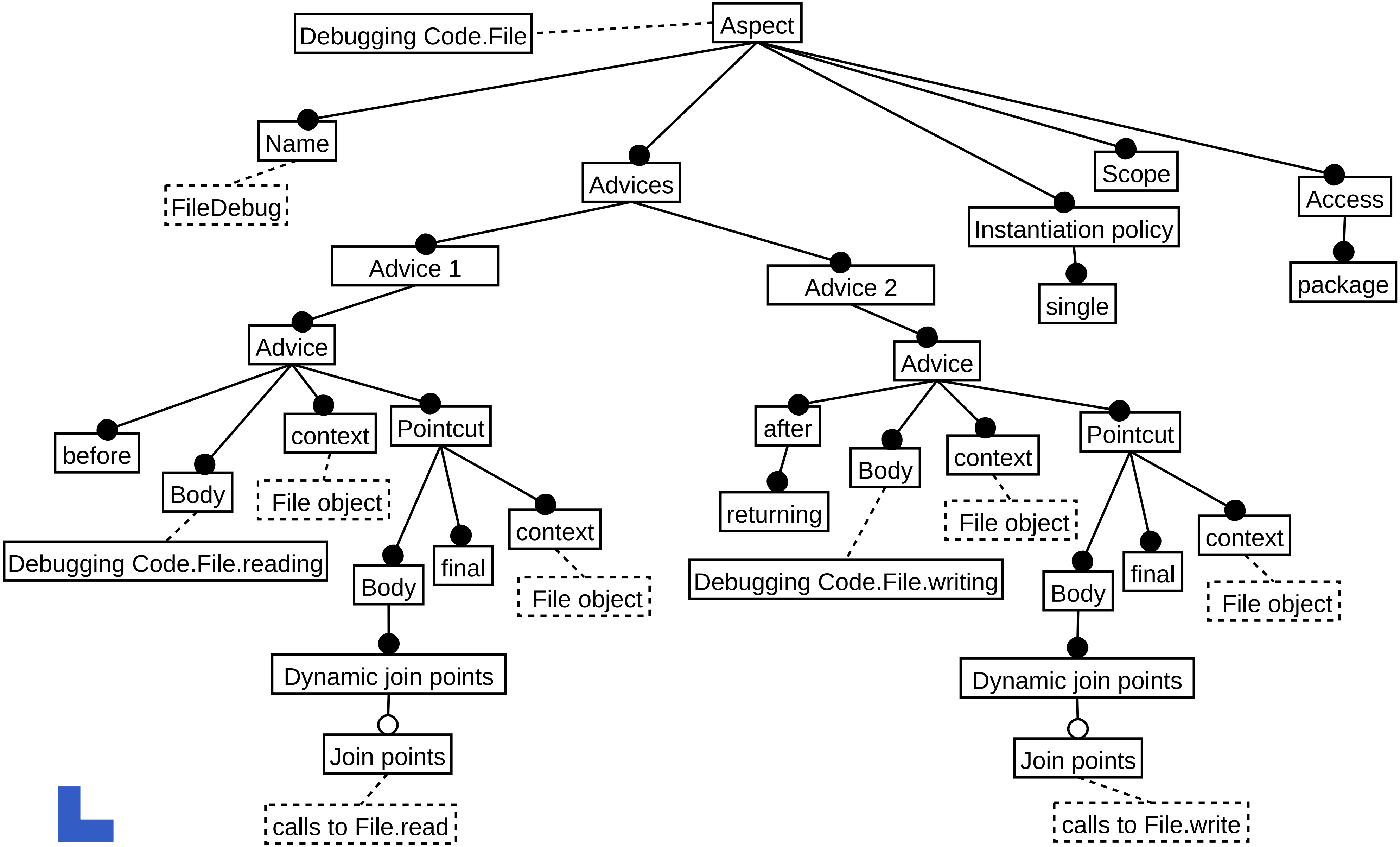


Transformational Analysis Example





Transformational Analysis Example



Code Skeleton Design

- Performed by traversing paradigm instances
- Structural paradigm instances first
- An example: the file debugging code aspect

```
aspect FileDC {
     before(File f):
         target(f) && call(* File.read(..)) {
             . . .
         }

     after(File f):
         target(f) && call(* File.write(..)) {
             . . .
         }
}
```

Aspect-Oriented Modeling and MPD$_{FM}$

- Aspect-oriented languages differ in essential aspect-oriented mechanisms
- Hard to generalize them for modeling purposes
- MPD$_{FM}$ application domain feature models
  - Abstract from any implementation mechanisms
  - Independent of a solution domain feature model
- AspectJ paradigm model enables to identify aspects early in the design

Summary (1)

- Multi-paradigm design with feature modeling (MPD_{FM}):
  - Both application and solution domain represented as feature models
  - Transformational analysis based fully on feature modeling
- MPD_{FM} for AspectJ
  - AspectJ paradigm model
  - Demonstrated in the text editing buffer transformational analysis
  - Successfully applied to the domain of feature modeling

Summary (2)

- Reuse of application and solution domain feature models
- Improvements of feature modeling:
  - Concept instantiation with respect to instantiation time
  - Parameterization in feature models
  - Constraints and default dependency rules as logical expressions
  - Concept references
  - A dot convention to enable referring to concepts and features unambiguously
  - A parameterized concept for representing cardinality in feature modeling

Further Work

- Partial feature model reuse
  - Overlapping domains
  - Generalization of similar concepts from different domains
- $\mathrm{MPD}_{\mathrm{FM}}$ specialization to solution domains other than programming languages