Modeling Object Oriented Systems via Controlled English Verbalization of Description Logic

CNL2010

Paweł Kapłański
Gdańsk University of Technology
Poland

CONTENT

- Software Engineering - The Problem
- Software Modeling - The Domain
- How to formalize OO software models
- Why Description Logic?
- Why CNL instead of OCL?
- My approach – what is done so far
- The Toolchain
- Possible future work
- Discussion

The Problem

- Software systems are more and more complex
  - Without a support of formal-methods it is almost impossible to trace and understand consequences of even small change of design in a complex software system. → Strategic decisions that are made by the authorities lack of information about the real state of the software product
- Software engineers are in general not familiar with formal methods:
  - They use natural language to describe constrains (in comments)

Software Modeling

- UML+OCL
  - UML is a standard software modeling language nowadays
  - Users of the UML can use OCL to specify constraints and other expressions attached to their models
- LePUS3
  - Capture and convey the building-blocks of object-oriented design
  - Automatic verification of Design Patterns
- SEMAT Initiative
  - **Ivar Jacobson,** Bertrand Meyer, and Richard Sole
  - Identifies "Methods&Tools" as one of macro-trends in modern Software Engineering
  - Back to the Clean-Room methodologies
  - Searching for a Kernel Language

SEMAT – Kernel language Requirements

- The language can cover all relevant practices and patterns, and their composition, in today's methods.
- It supports composing them in different ways to describe new method elements.
- It is extendible, allowing the description of yet-to-be invented method elements and their elements (such as individual practices).
- The descriptions are easy to understand; the language should be designed for the developer community (not just process engineers and academics).
- The language support simulating the application of method elements.
- The language provide validation mechanisms.

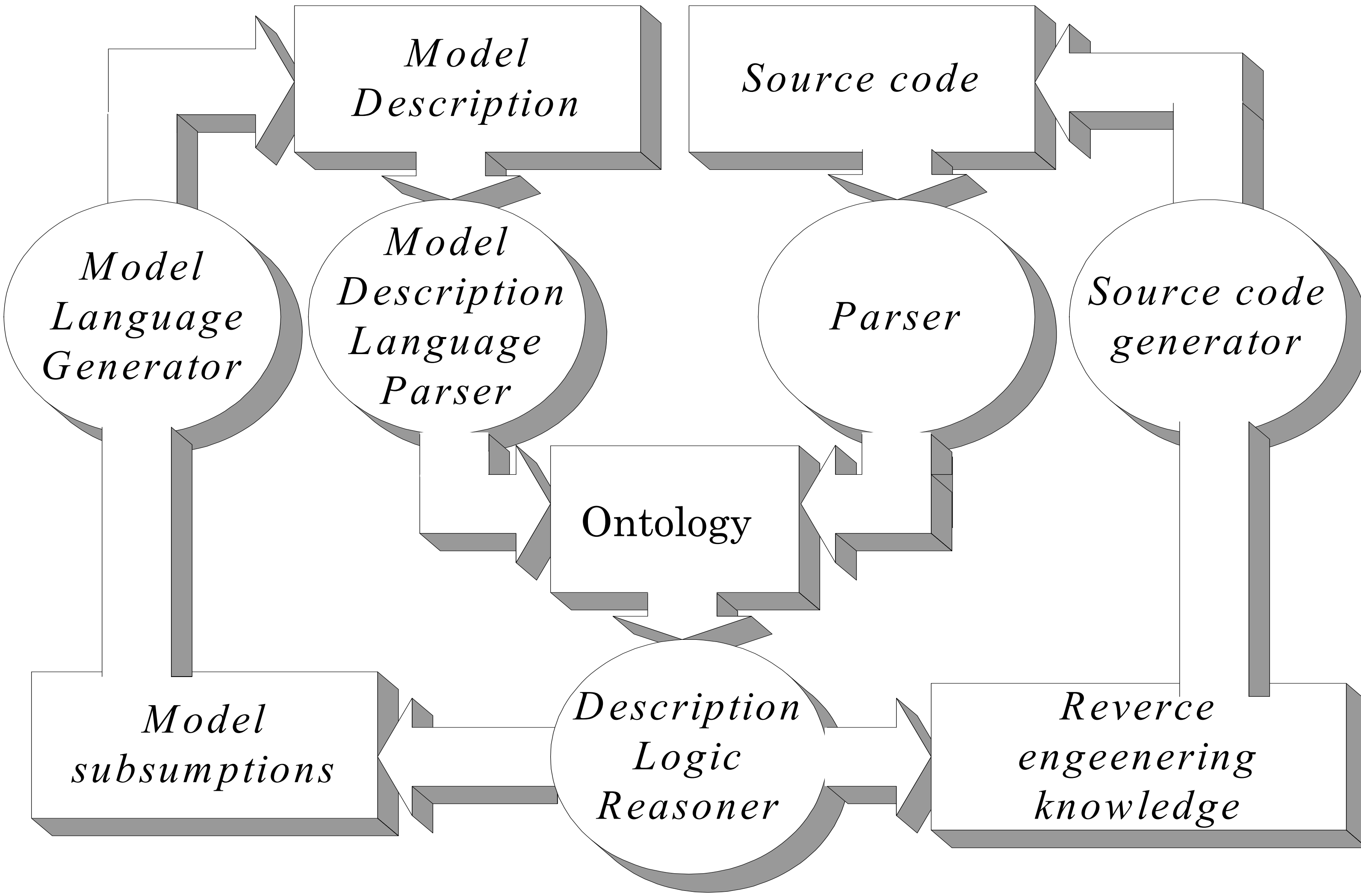
The Domain

- What is a static structure (or a code structure) of a computer program?
  - Opposite to runtime-structure
  - Well defined in terms of compilation-time to machine code
  - Not well defined from theoretical perspective
- Why is it important?
  - Separates the computer program to design and implementation
  - The software design can be treated as an ontology
  - Shifts imperative parts to declarative ones

Description Logic

- Mostly used in Semantic-Web
  - The math behind OWL
- DL is decidable fragment of FOL (with additional features)
  - It is a core property for static structures
- Complexity vs. Responsiveness
  - Effective reasoning implemented
- It is possible to Verbalize it in CNL
  - ACE ⟷ OWL
  - CE_{DL} –prototypical implementation in JavaCC

THE TOOLCHAIN



DL Constructors

	Construction	Meaning	Transcription
[illegible]	C,D → A	*atomic concept*	<noun>
	$\top$	*universal concept*	**something**
	$\bot$	*bottom concept*	**nothing**
	$\neg$A	*atomic negation*	**not** <noun>
	C $\sqcap$ D	*intersection*	… **and** ...
	$\forall$R.C	*value restriction*	<verb> **only** <noun>
	$\exists$R.$\top$	*weak existential restriction*	<verb> **something**
[illegible]	C $\sqcup$ D	*union*	… **or** ...
[illegible]	$\exists$R.C	*existencial restriction*	<verb> <noun>
[illegible]	$\neg$ C	*negation*	**not** ...
[illegible]	$\leq$ nR	*at most*	<verb> **at most** <nunmer> things
	$\geq$ nR	*at least*	<verb> **at least** <number> things
[illegible]	{a1, a2,…, an}	*one of*	**either** a1 **or** a2 … an
[illegible]	P,Q,R → R^{-}	*inverse relation*	<verb> **by**
[illegible]	$\leq$ nR.C	*quantified*	<verb> **at most** <nunmer> <noun>
	$\geq$ nR.C	*number restriction*	<verb> **at least** <number> <noun>
[illegible]	P $\circ$ Q $\sqsubseteq$ R	*complex role inclusion*	**if** <role expr>

DL Expressions

Construction	Meaning	Transcription
$C \sqsubseteq D$	*concept*	**Every** <C> **is a** <D> .
$C \sqsubseteq \exists R . D$	*specification*	**Every** <C> <R> **a** <D>.
$C(I)$	*assertions*	<I> **is** <C>.
$R(I,J)$		<I> <R> <J>.
$C(x)$, $\{x\} \sqsubseteq D$	*unnamed*	**The** <C> **is a** <D>.
$D(x)$, $C(x)$	*instances*	**The** <C> **is the** <D>.

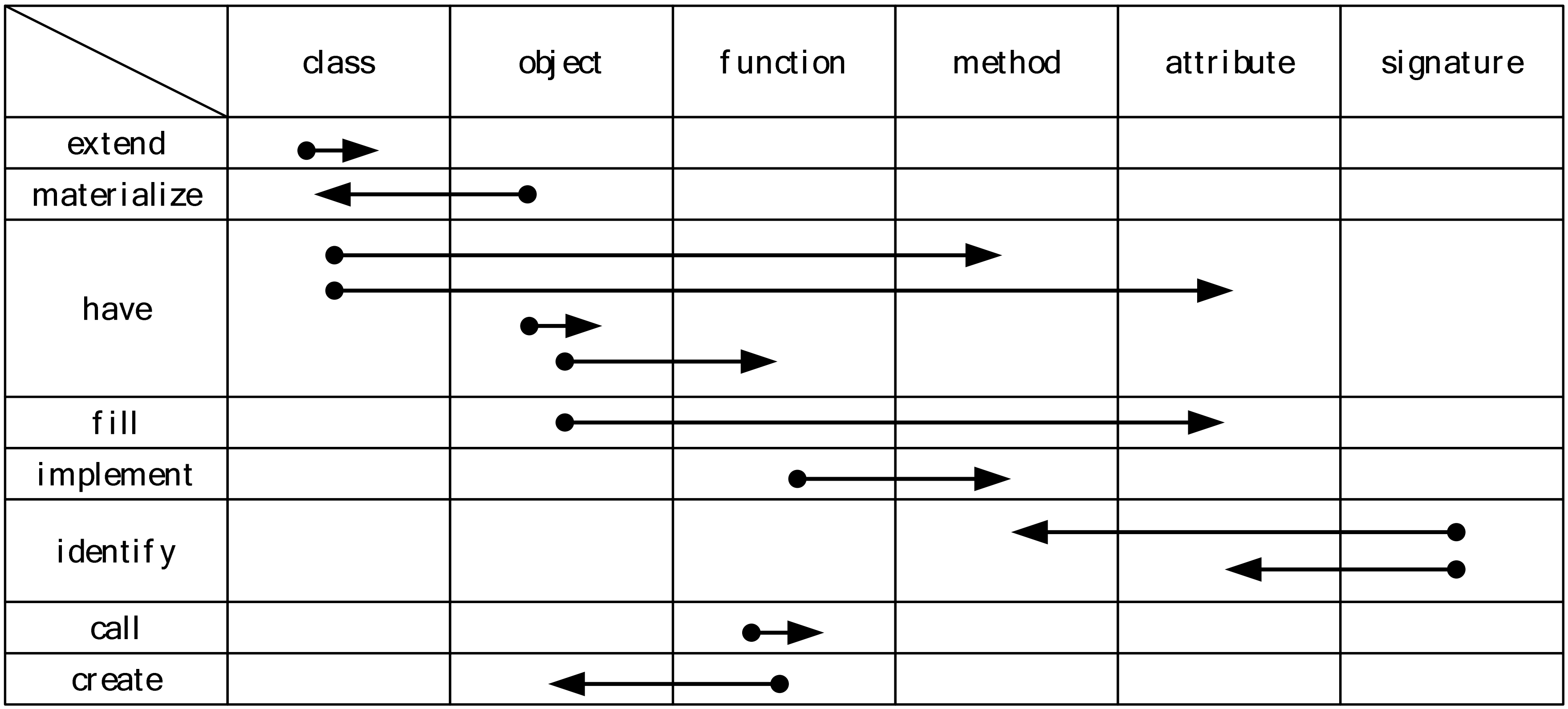
The semantic of OWL and UML are different

- How to map UML to OWL
  - UML classes are like templates that define a runtime object and its storage.
  - OWL classes are like labels for concepts,
  - There is no 1:1 full mapping (problems include aggregation and polymorphism)
  - There is no way to describe high-level structures
- Diego Calvanese (2005): Mapping UML to *ALUNI*.
  - Reasoning on UML class diagrams
- OMG
  - Metamodels of UML and OWL

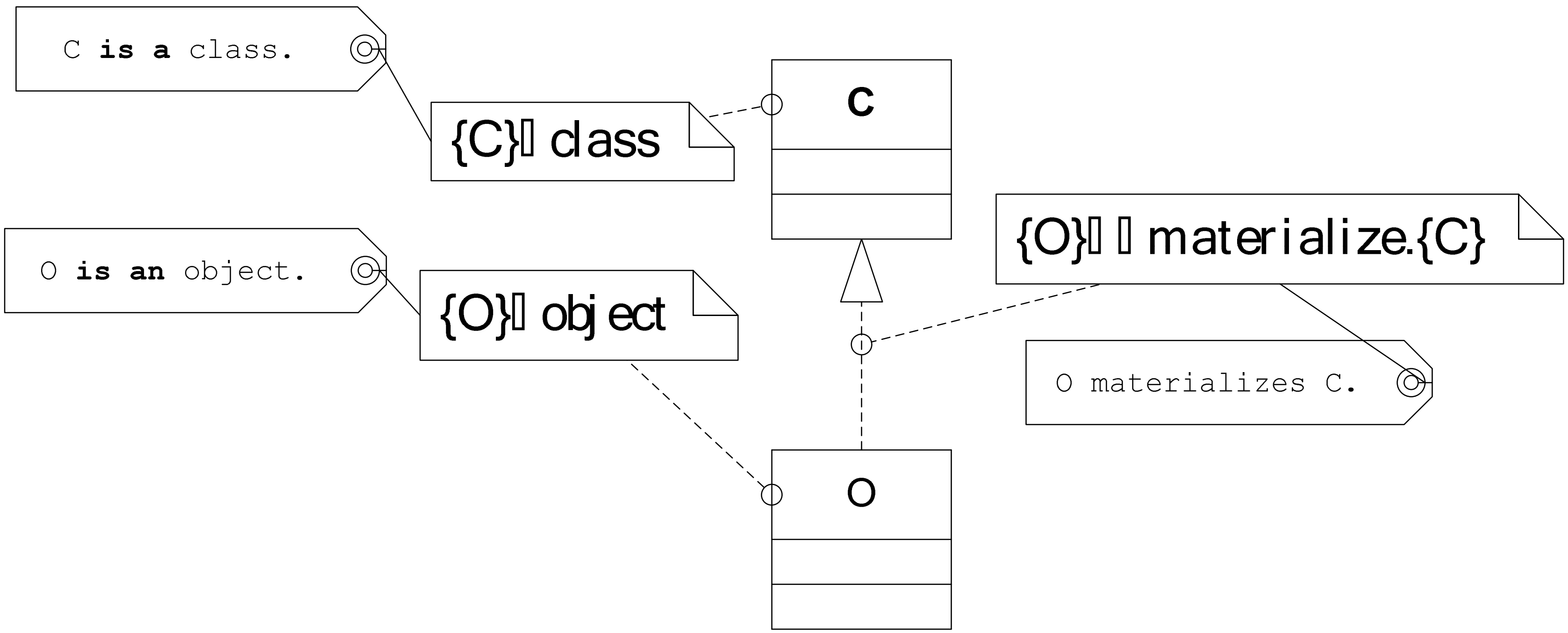
The Alternative Mapping

- **Class as an instance**
- it can be threaded as an instance in fact – in some OO languages class is an runtime object - reflexion
- We can build complex expressions in DL for classes

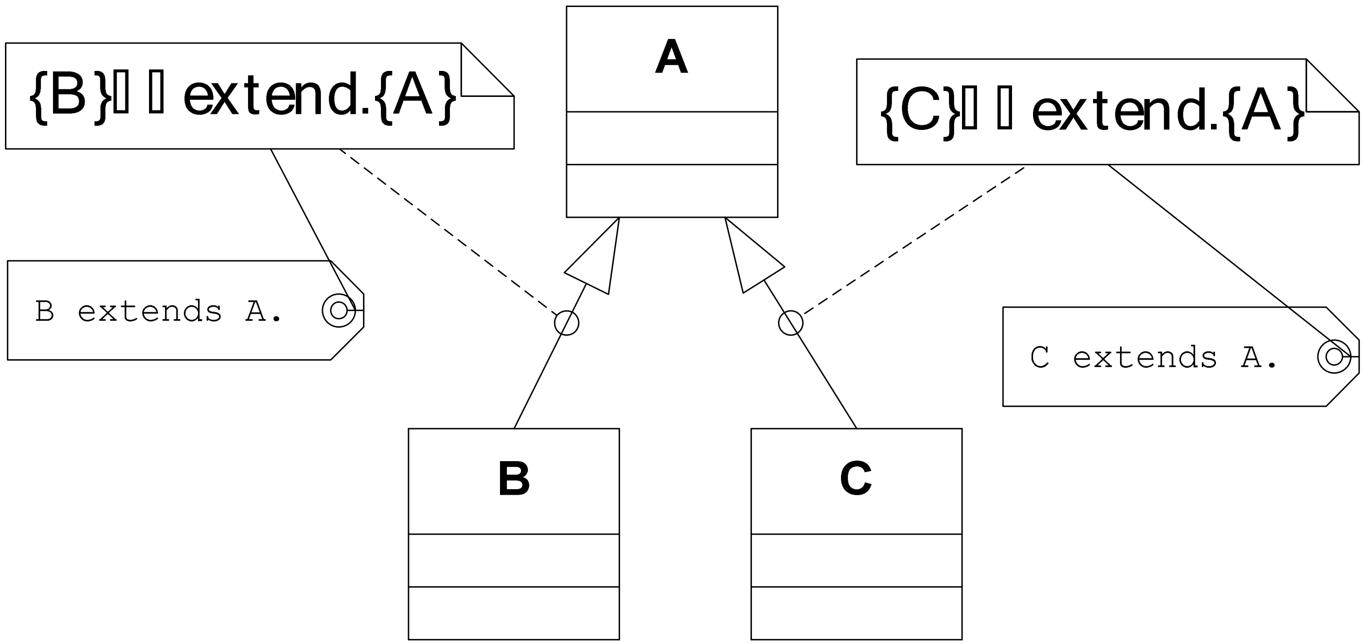
Basic OO Model
Domains and Ranges



CLASS AND OBJECT



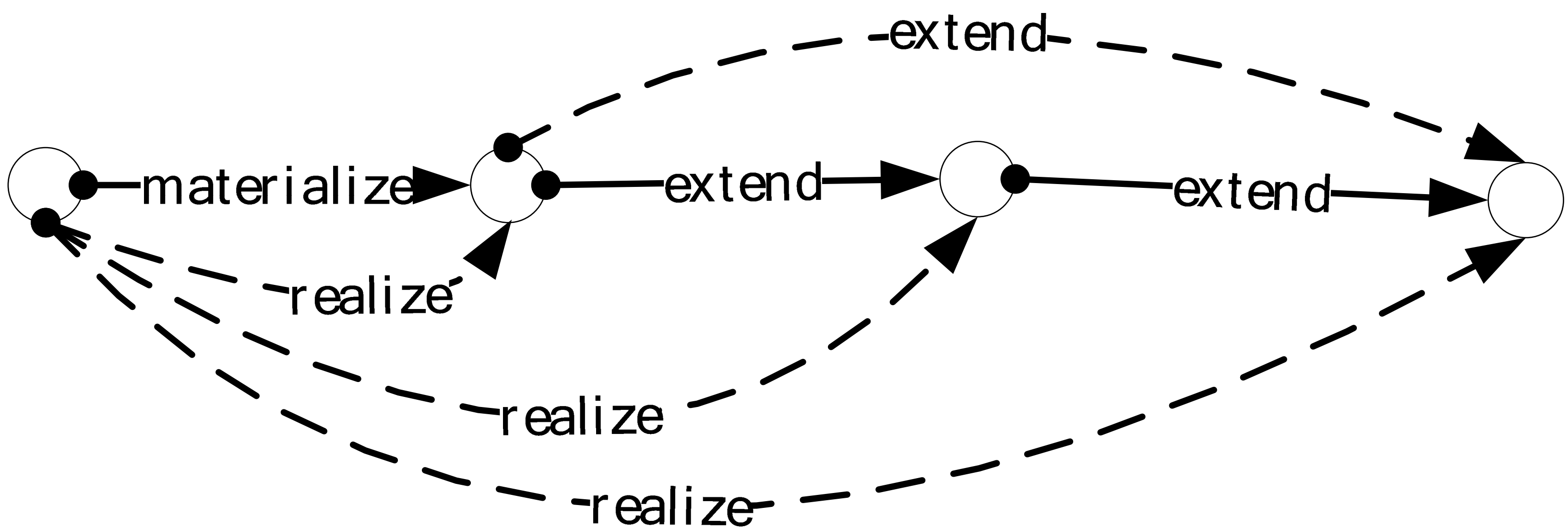
Class Inheritance



Properties Of Roles

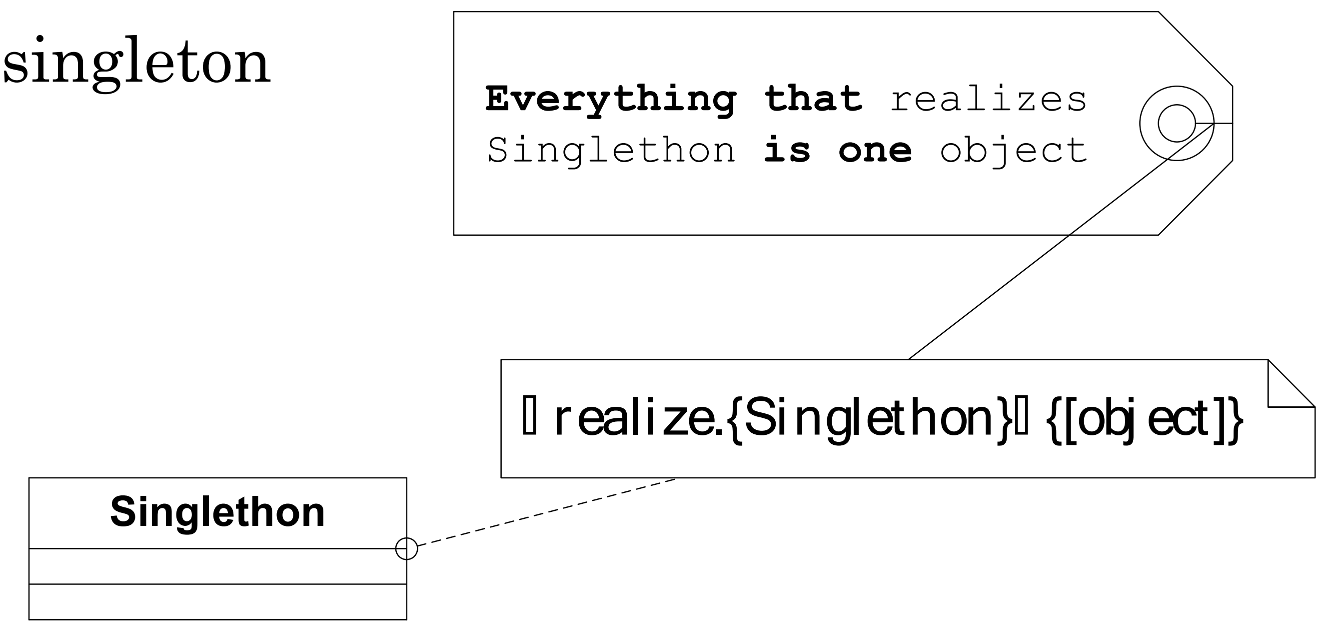
materializemrealize,
materializeéextendmrealize

materialize(éextend)* mrealize



SIMPLE CONSTRAINS

- Abstract class → $realize.^
- Final class → $extend.^
- The singleton

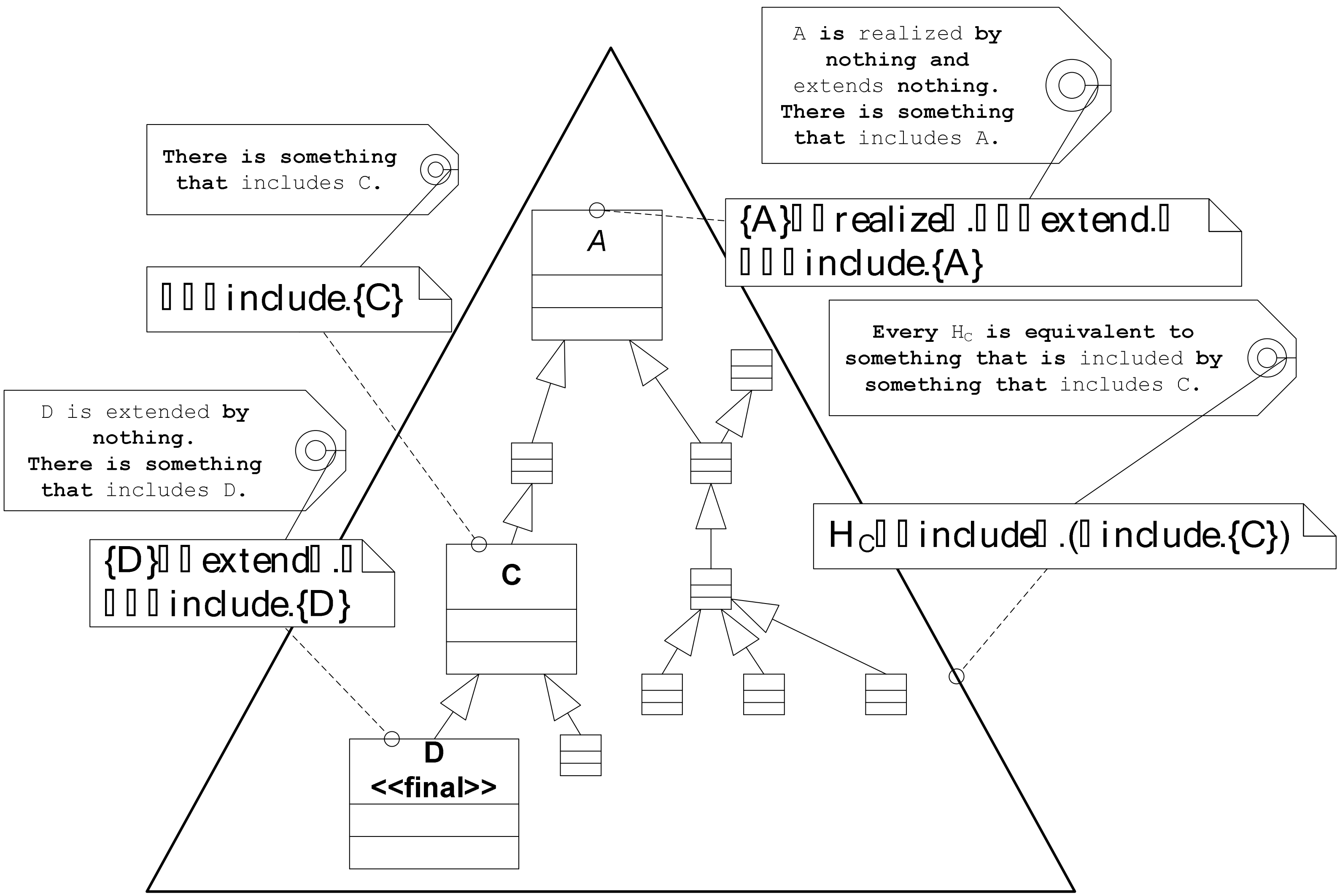


HIERACHIES

- Hierarchy of classes is a set that contain all classes related with “extend” relationship
- If *C* is a class

  - ¨m$include.{C}
  - includeéextendminclude,
  - includeéextend⁻minclude
  - H_Cº $include⁻.($include.{C}).

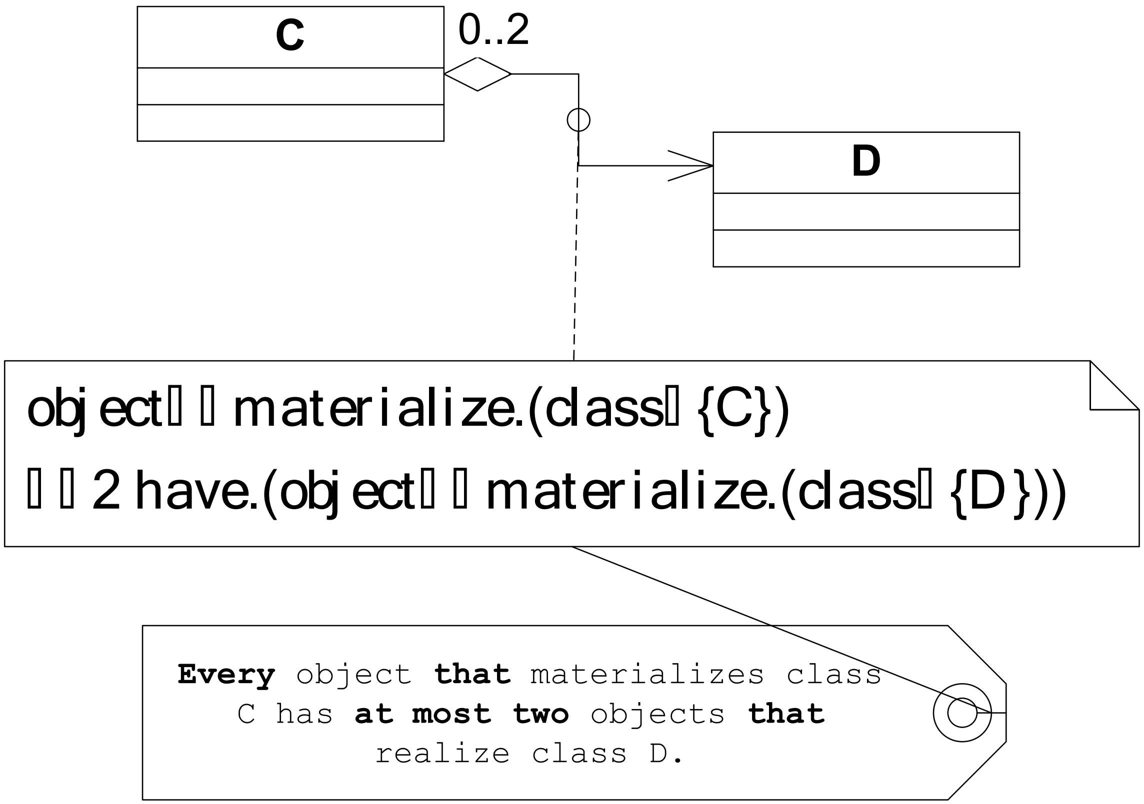
Class hierarchy



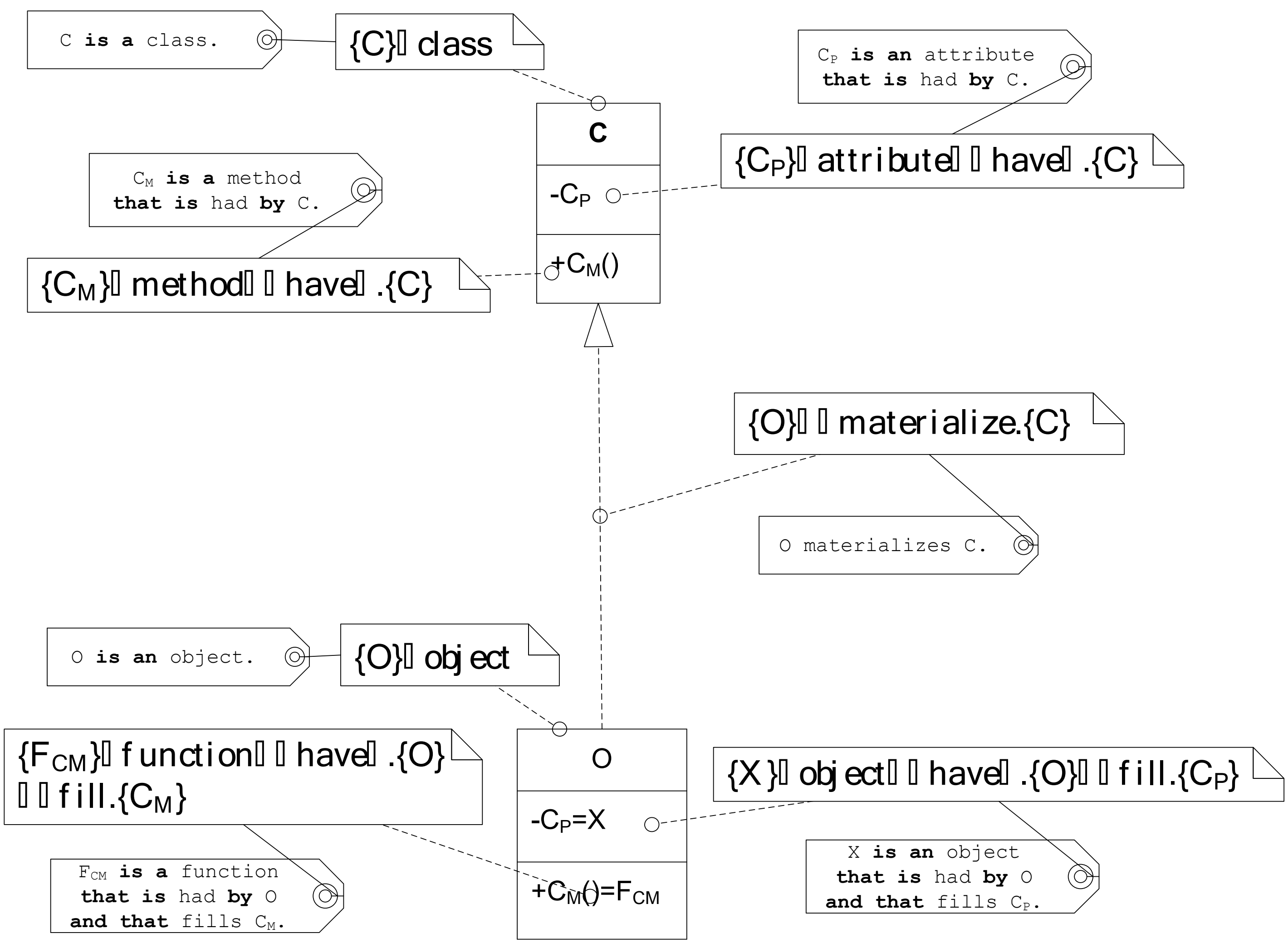
THE MODEL OF PART-WHOLE

- Modeled by a “have” role
  - extendéhavemhave
  - Implies that have^{-}éextend^{-}mhave^{-}
  - if between two classes C and D exists chain of “extend” roles: {C}m$extend^{-}.$extend^{-}....$extend^{-}.{D}
  - then it can be inferred that: $have^{-}.{C}m$have^{-}.{D}
  - That means that everything that is had by C is had also by D
  - **It can be interpreted as inheritance**

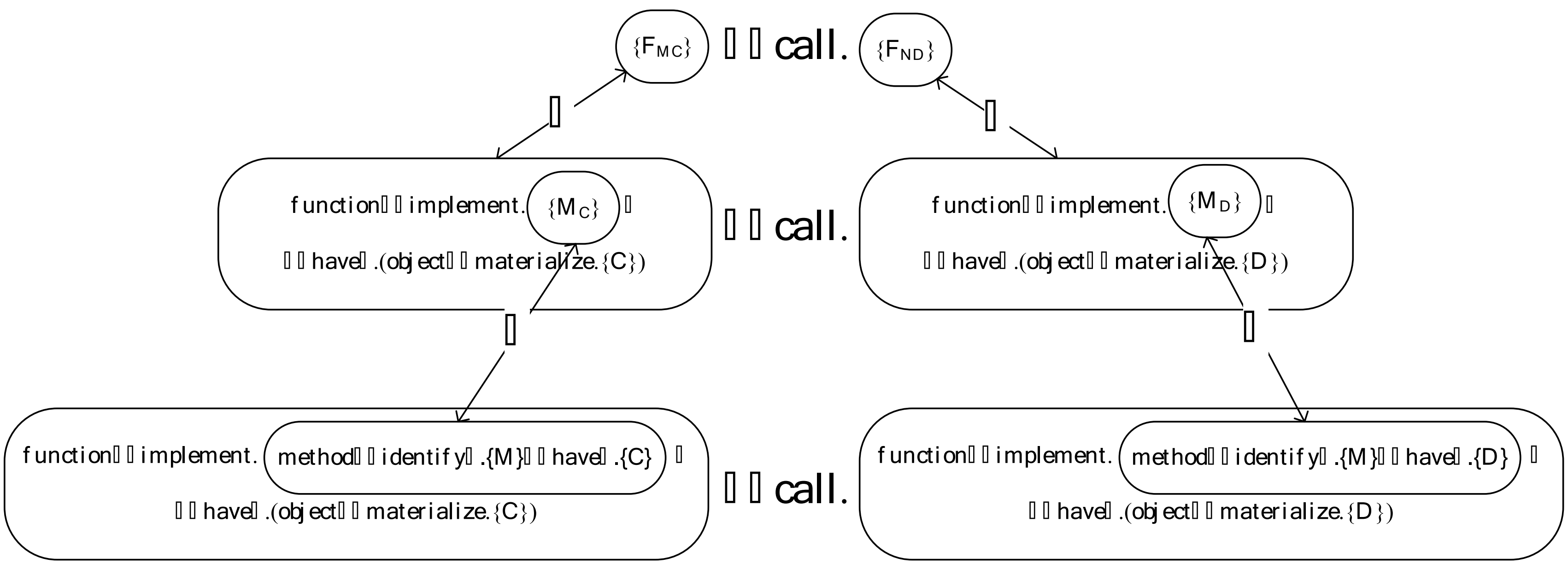
The aggregation and its representation



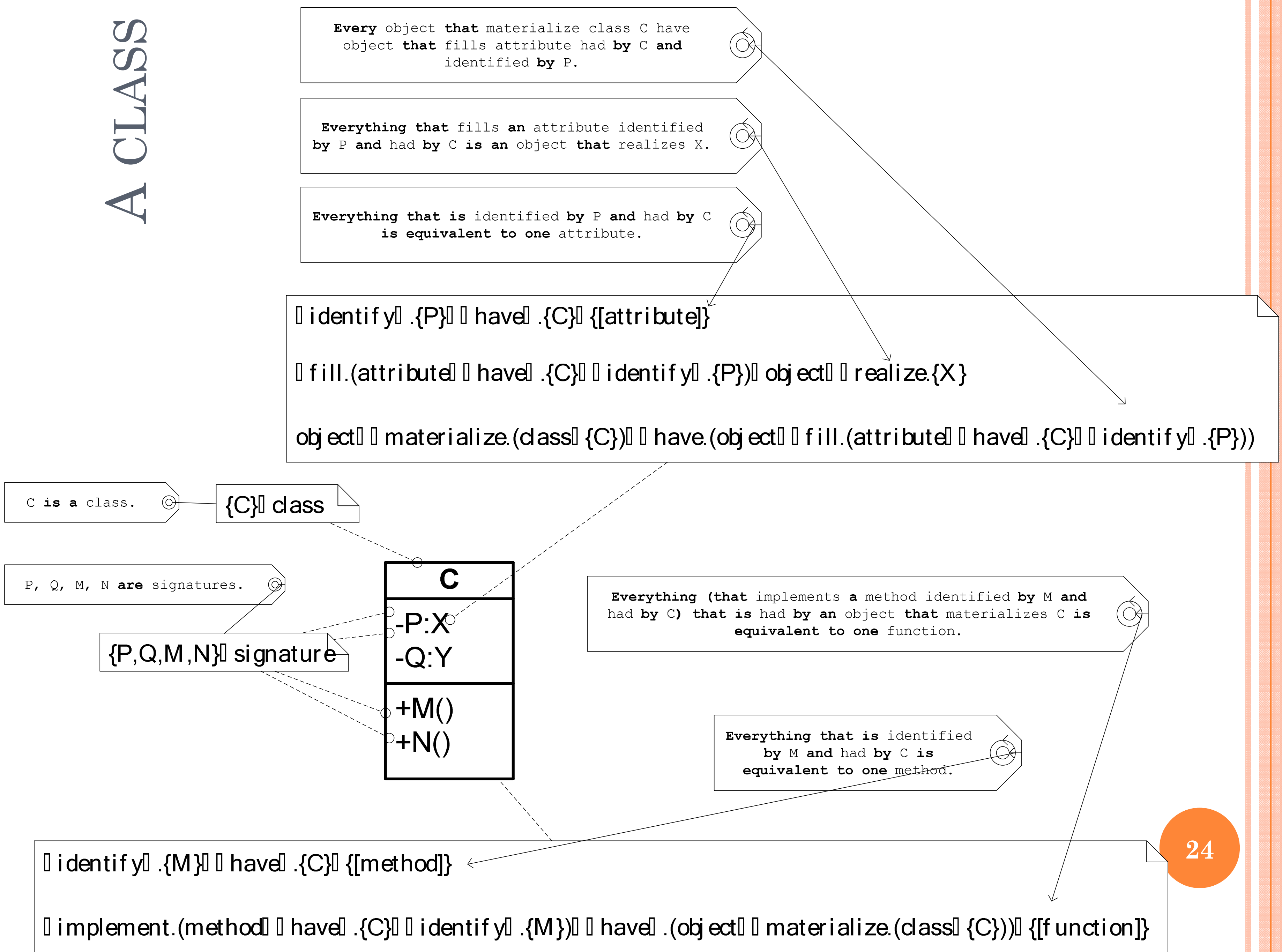
PARTS OF CLASS AND PARTS OF OBJECT



EXTENDING FUNCTION NAMES TO SUPERIMPOSITIONS OF SIGNATURES AND CLASSES

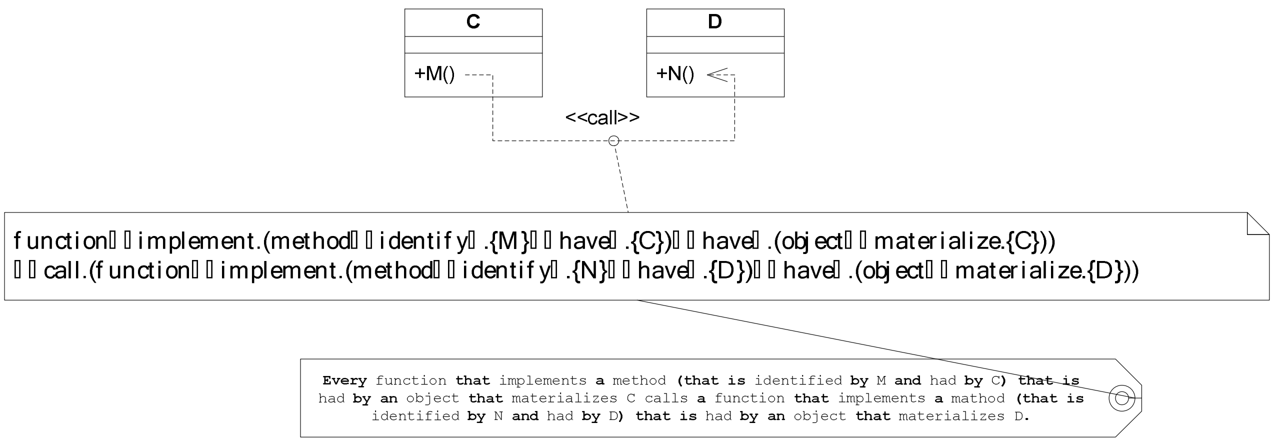


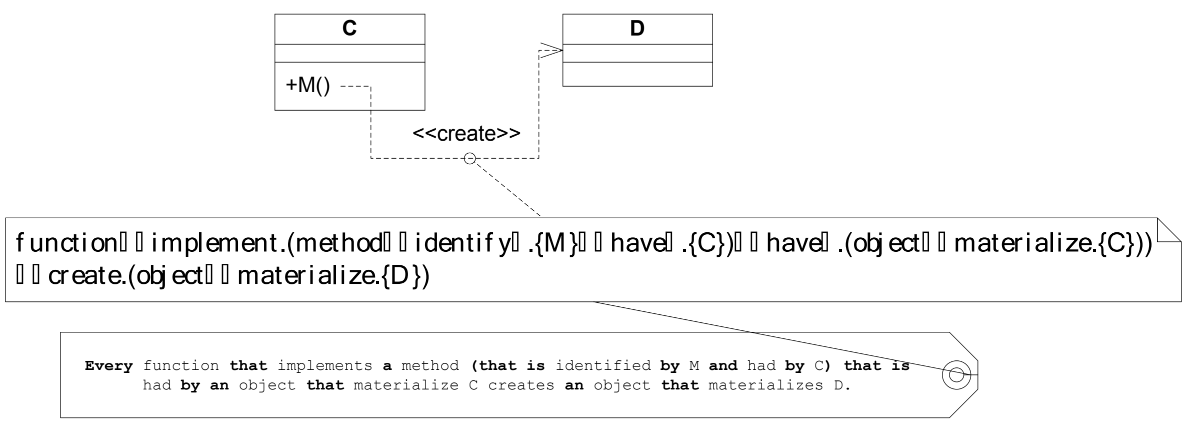
A CLASS



Every object that materialize class C have object that fills attribute had by C and identified by P.
Everything that fills an attribute identified by P and had by C is an object that realizes X.
Everything that is identified by P and had by C is equivalent to one attribute.
□ identify□ .{P}□ □ have□ .{C}□ {[attribute]}
□ fill.(attribute□ □ have□ .{C}□ □ identify□ .{P})□ object□ □ realize.{X}
object□ □ materialize.(class□ {C})□ □ have.(object□ □ fill.(attribute□ □ have□ .{C}□ □ identify□ .{P}))
C is a class.
{C}□ class
P, Q, M, N are signatures.
{P,Q,M,N}□ signature
C
-P:X
-Q:Y
+M()
+N()
Everything (that implements a method identified by M and had by C) that is had by an object that materializes C is equivalent to one function.
Everything that is identified by M and had by C is equivalent to one method.
□ identify□ .{M}□ □ have□ .{C}□ {[method]}
□ implement.(method□ □ have□ .{C}□ □ identify□ .{M})□ □ have□ .(object□ □ materialize.(class□ {C}))□ {[function]}

CALL AND CREATE





The Practical Problems

- Inconsistencies between requirements, system design, test cases and the source code that appear in the process of software development
- Violations of design constraints and the existing architectural style found in the source code
- The effects of changes on the reliability (traceability of the system) of the system and analyze the risks of a change.

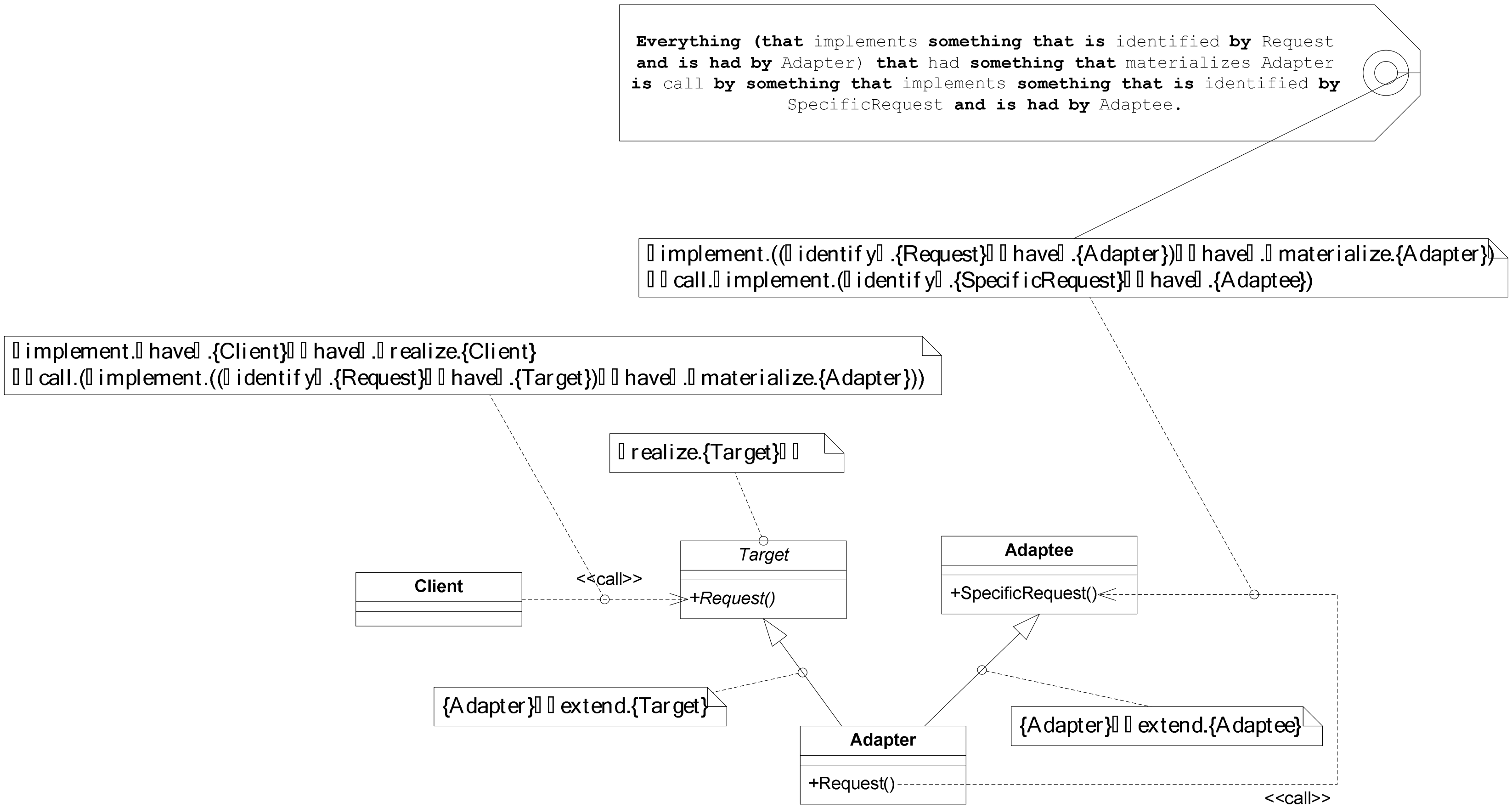
Design Patterns

- Why we want to formalize Design Pattern?
  - We can detect them in existing code
  - We can check if there are no violations of DP
  - We can reuse them – they become language independent
  - With CNL we can use the natural language as a design-pattern language

ADAPTER

- Adapter design pattern (often referred to as the wrapper pattern or simply a wrapper) is a design pattern that translates one interface for a class into a compatible interface. An adapter allows classes to work together that normally could not because of incompatible interfaces, by providing its interface to clients while using the original interface. The adapter translates calls to its interface into calls to the original interface, and the amount of code necessary to do this is typically small (...)

ADAPTER


Everything (that implements something that is identified by Request and is had by Adapter) that had something that materializes Adapter is call by something that implements something that is identified by SpecificRequest and is had by Adaptee.
implement.((identify.{Request} have.{Adapter}) have.materialize.{Adapter})
call.implement.(identify.{SpecificRequest} have.{Adaptee})
implement.have.{Client} have.realize.{Client}
call.(implement.((identify.{Request} have.{Target}) have.materialize.{Adapter}))
realize.{Target}
Target
+Request()
Client
<<call>>
Adaptee
+SpecificRequest()
{Adapter} extend.{Target}
{Adapter} extend.{Adaptee}
Adapter
+Request()
<<call>>

Future Work

- The Tool – Ontology Aided Software Engeenering (OASE)

- Other Ontologies in SW:
  - Requirement specification
  - Run time testing - contracts

Discussion

- …