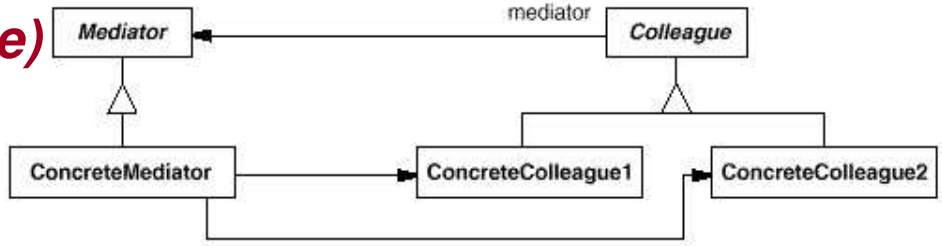
Static Extraction of Sound Hierarchical Object Graphs

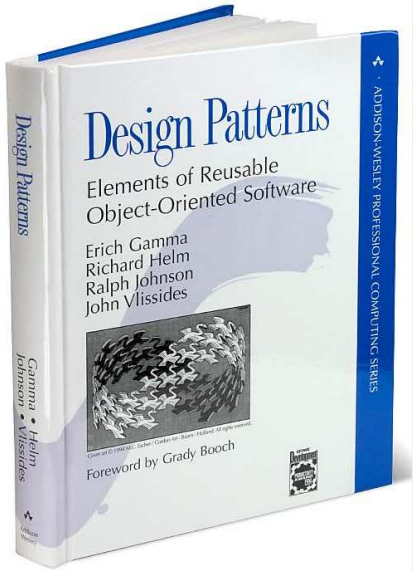
Marwan Abi-Antoun Jonathan Aldrich
School of Computer Science
Carnegie Mellon University

4th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)
January 24, 2009, Savannah, Georgia, USA

Structure

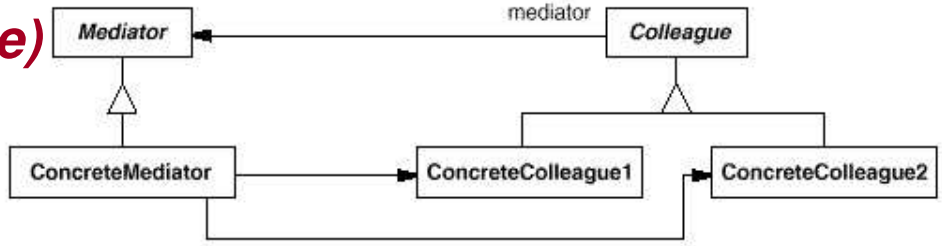
Class Diagram (Type structure)





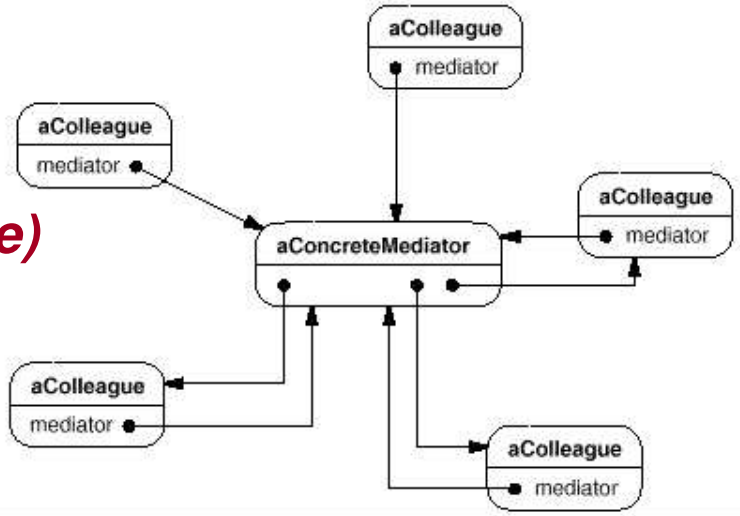
Structure

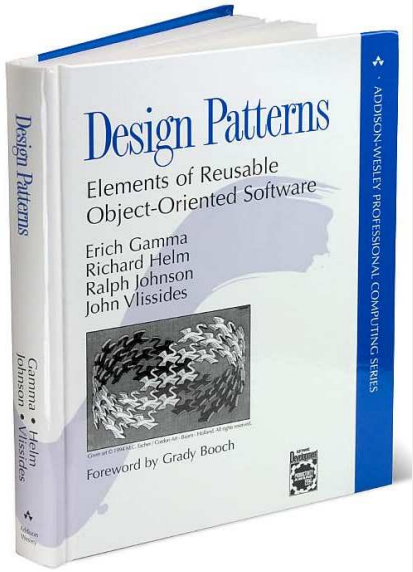
Class Diagram (Type structure)



A typical object structure might look like this:

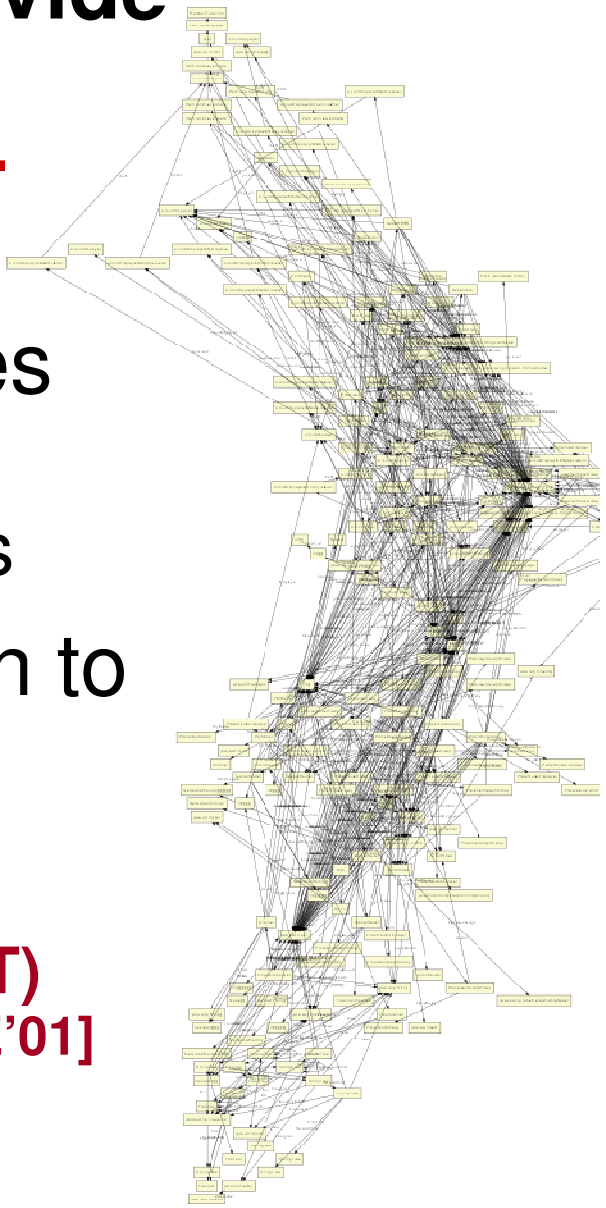
Object diagram (Instance structure)





Flat object graphs do not provide architectural abstraction

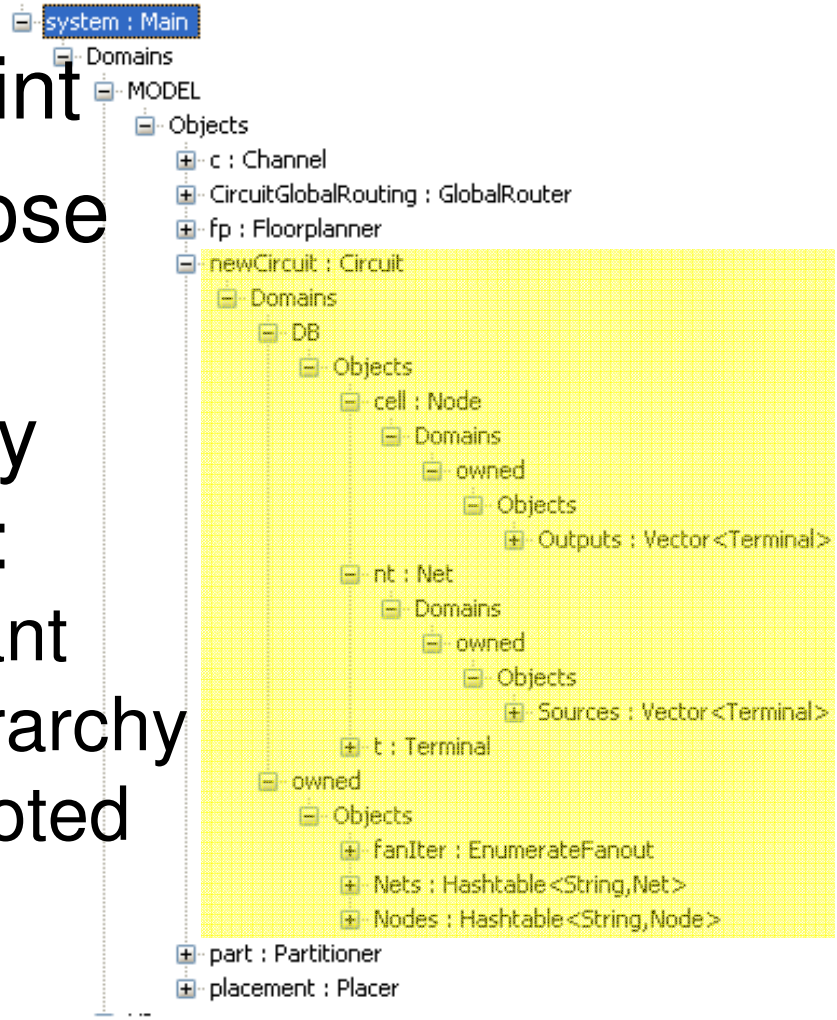
- Low-level objects mixed with architecturally significant ones
  - Show plethora of objects
  - No scale-up to large programs
- Require graph summarization to get readability [Mitchell, ECOOP'06]



Output of WOMBLE (MIT)
[Jackson and Waingold, TSE'01]
on 8,000-line system.

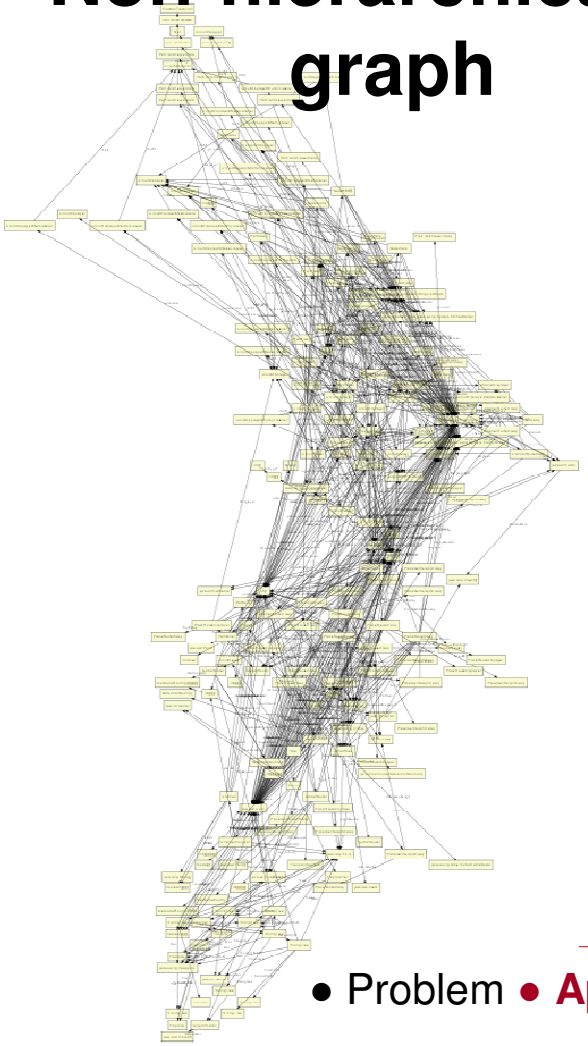
Use hierarchy to convey architectural abstraction

- Pick top-level entry point
- Use ownership to impose conceptual hierarchy
- Convey **abstraction** by **ownership hierarchy**:
  - Architecturally significant objects near top of hierarchy
  - Low-level objects demoted further down

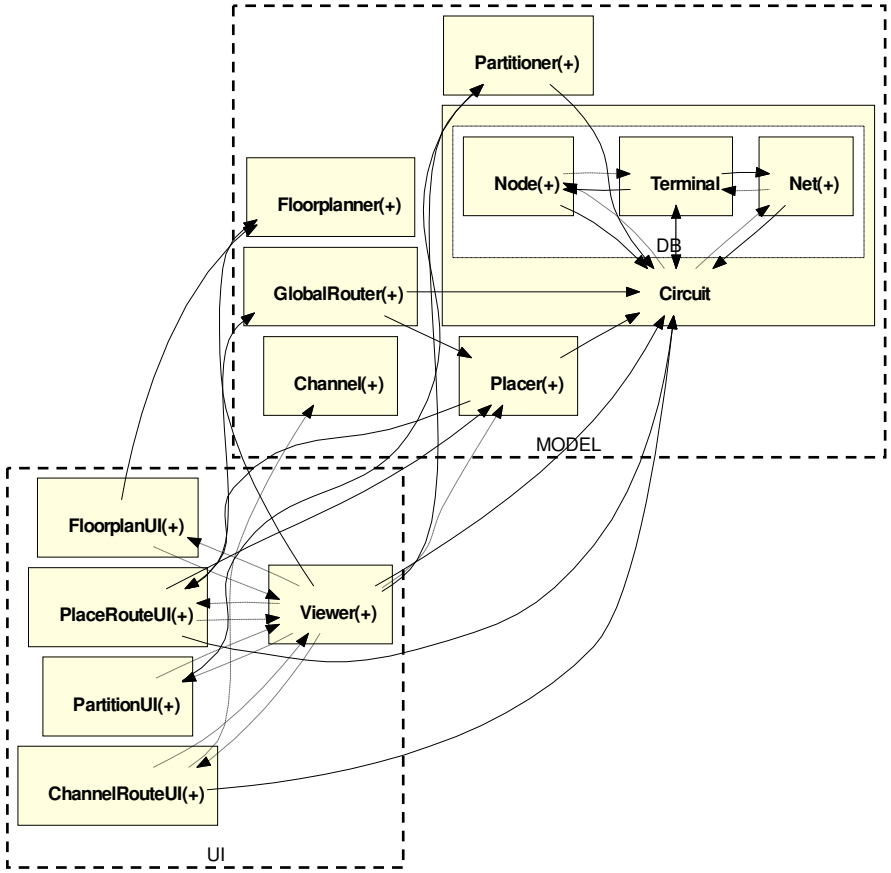


Collapse objects based on ownership (and types) to achieve abstraction

Non-hierarchical graph



Hierarchical graph



Central difficulty

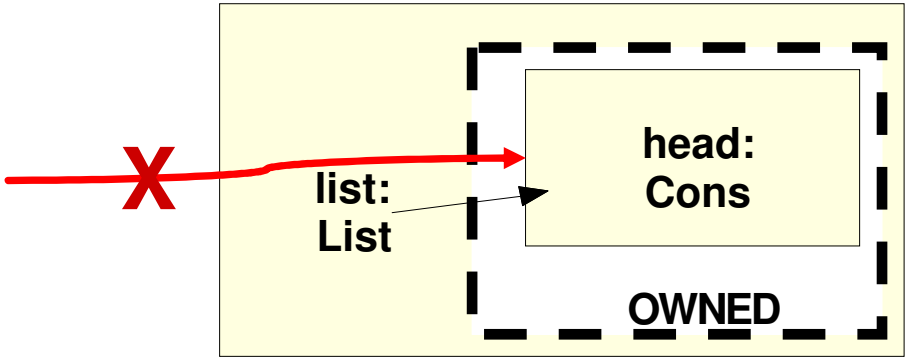
Architectural hierarchy not readily observable in program written in general purpose programming language

Core result

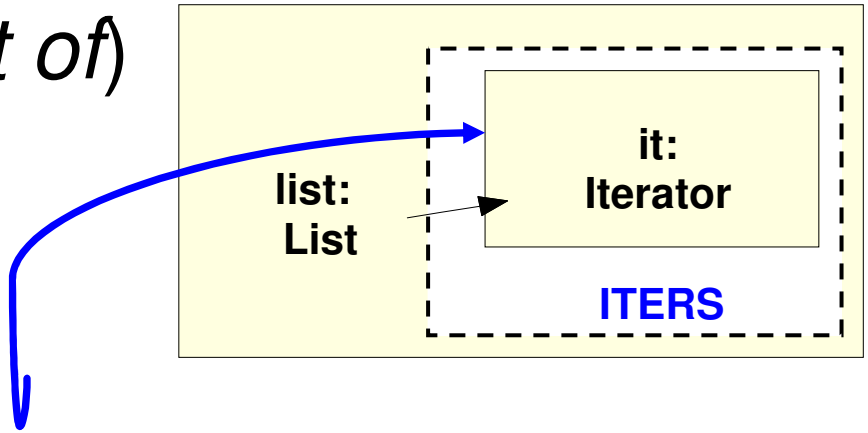
Leverage ownership type annotations to extract **hierarchical** object graph using **static analysis**

Annotations specify design intent

- Strict encapsulation (owned by)



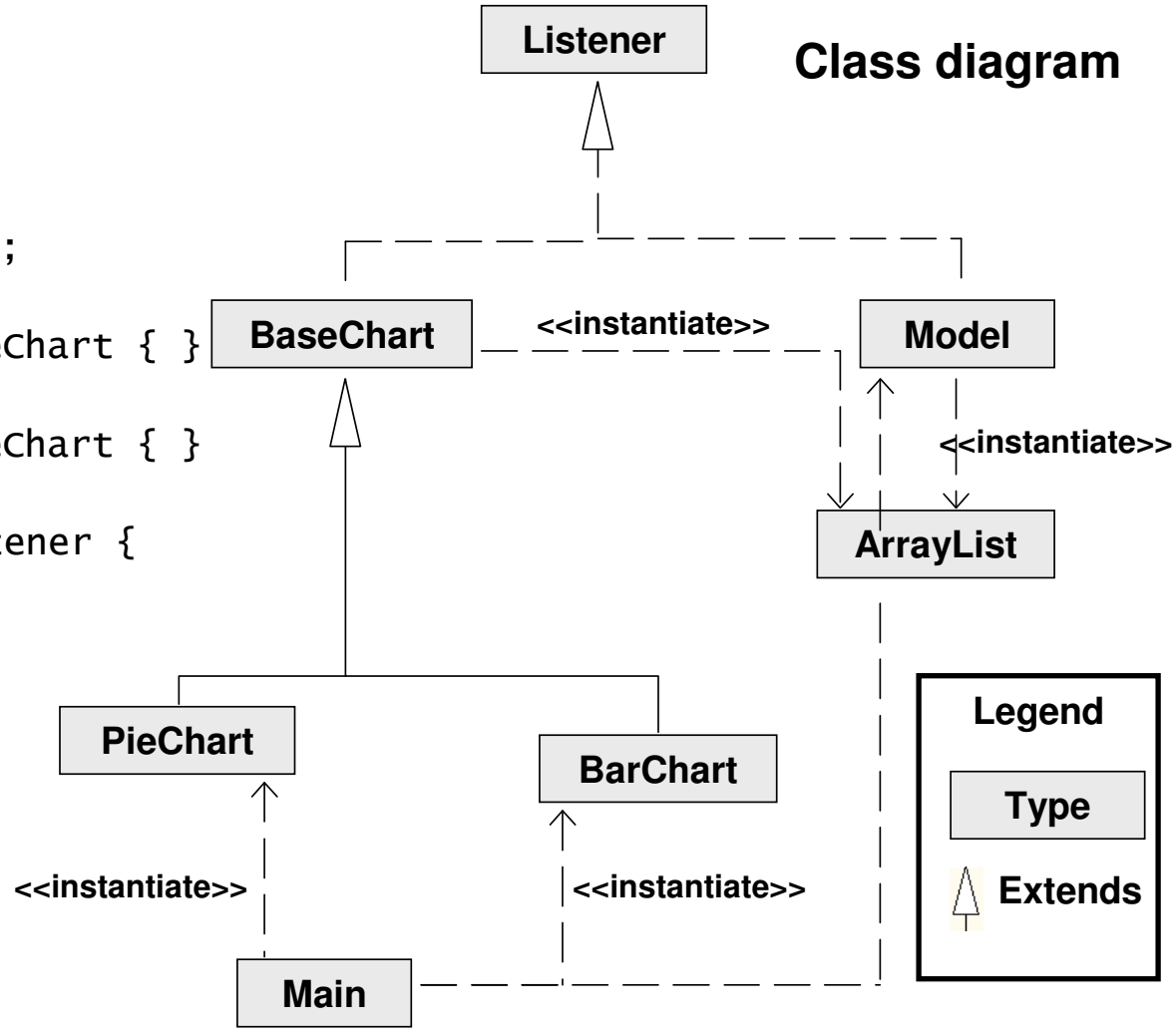
- Logical containment (conceptually *part of*)



Example: Listener system

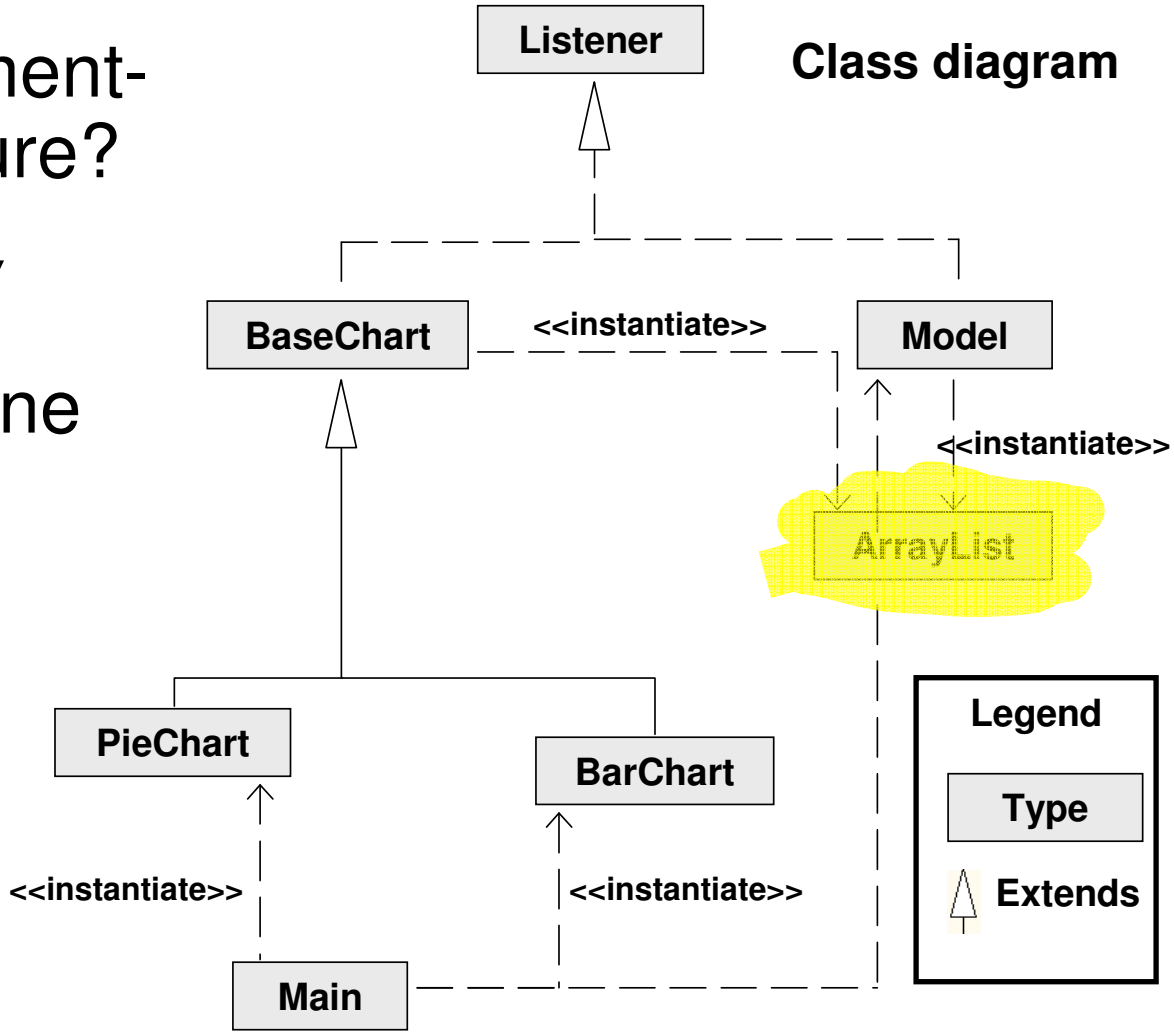
Listeners are hard to understand

```
interface Listener { }

class BaseChart
      implements Listener {
  List< Listener> listeners;
}
class BarChart extends BaseChart { }

class PieChart extends BaseChart { }

class Model implements Listener {
  List<Listener> listeners;
 }

class Main {
  Model model;
  BarChart barChart;
  PieChart pieChart;
}
```



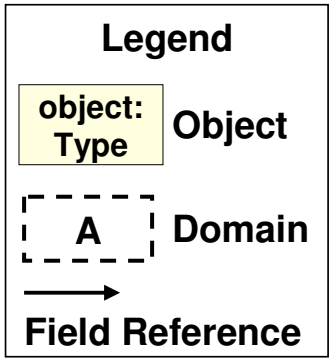
Class diagram leaves unanswered…

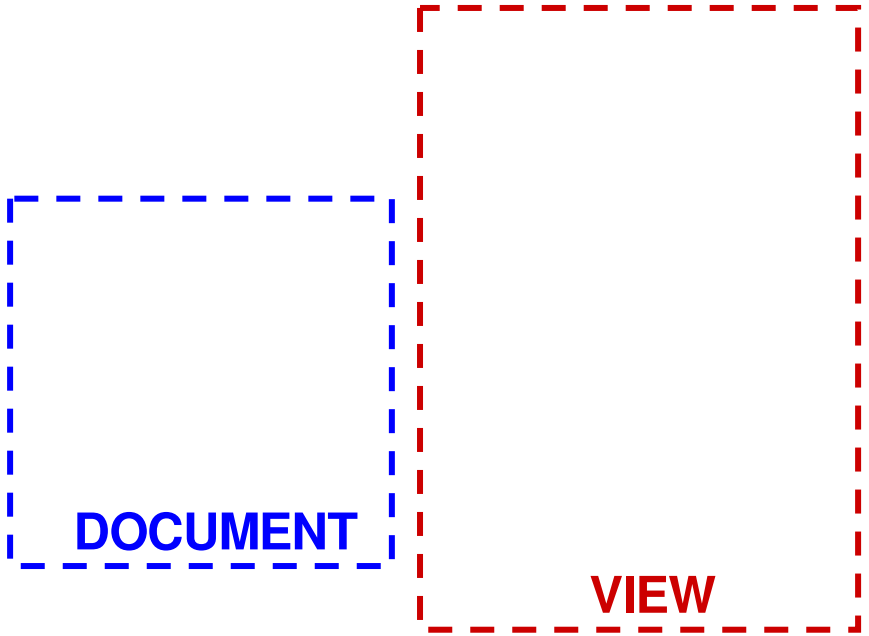
- Is this a Document-View architecture?
- Do `PieChart`, `BarChart`, `Model` share one `Listener`?
- Are different `ArrayList` instances conceptually different?



Object graph complements information in class diagram

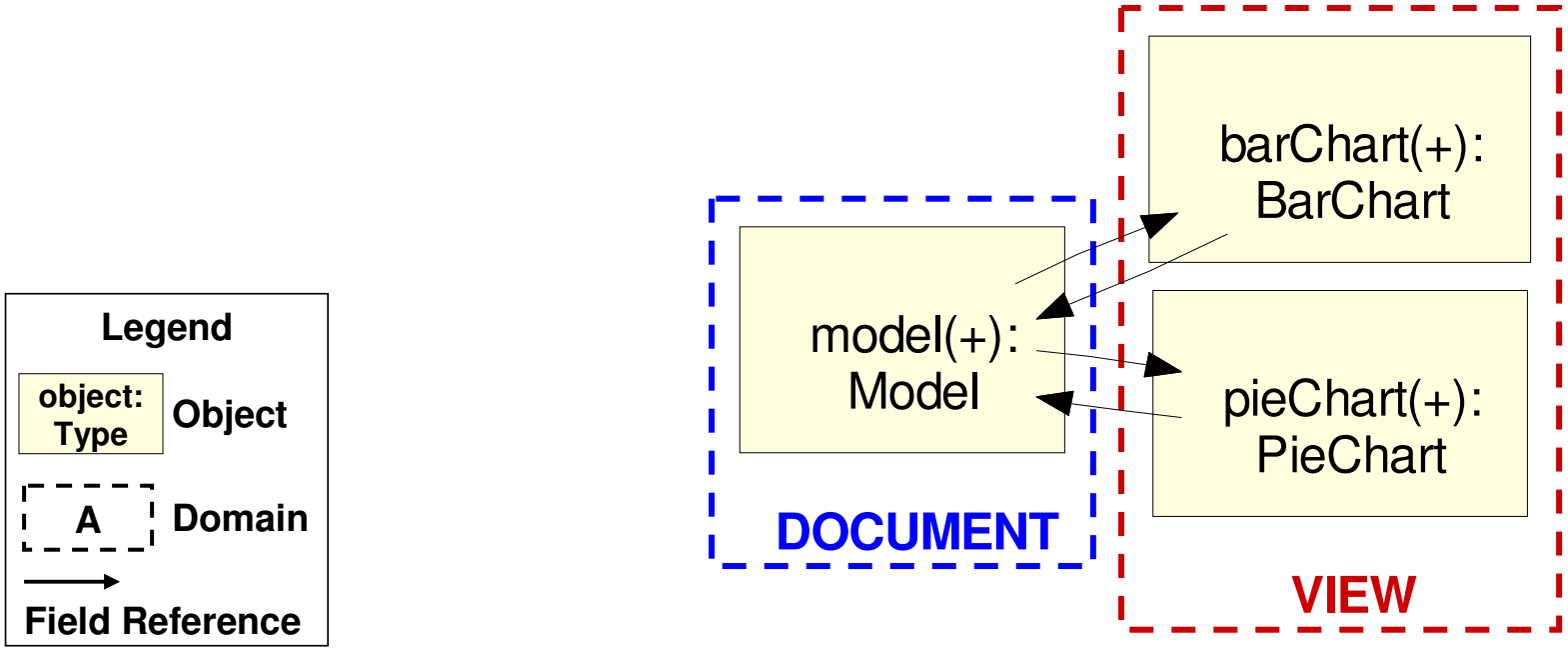
- **Domain** = **conceptual** group of objects
  - E.g., Document-View architecture





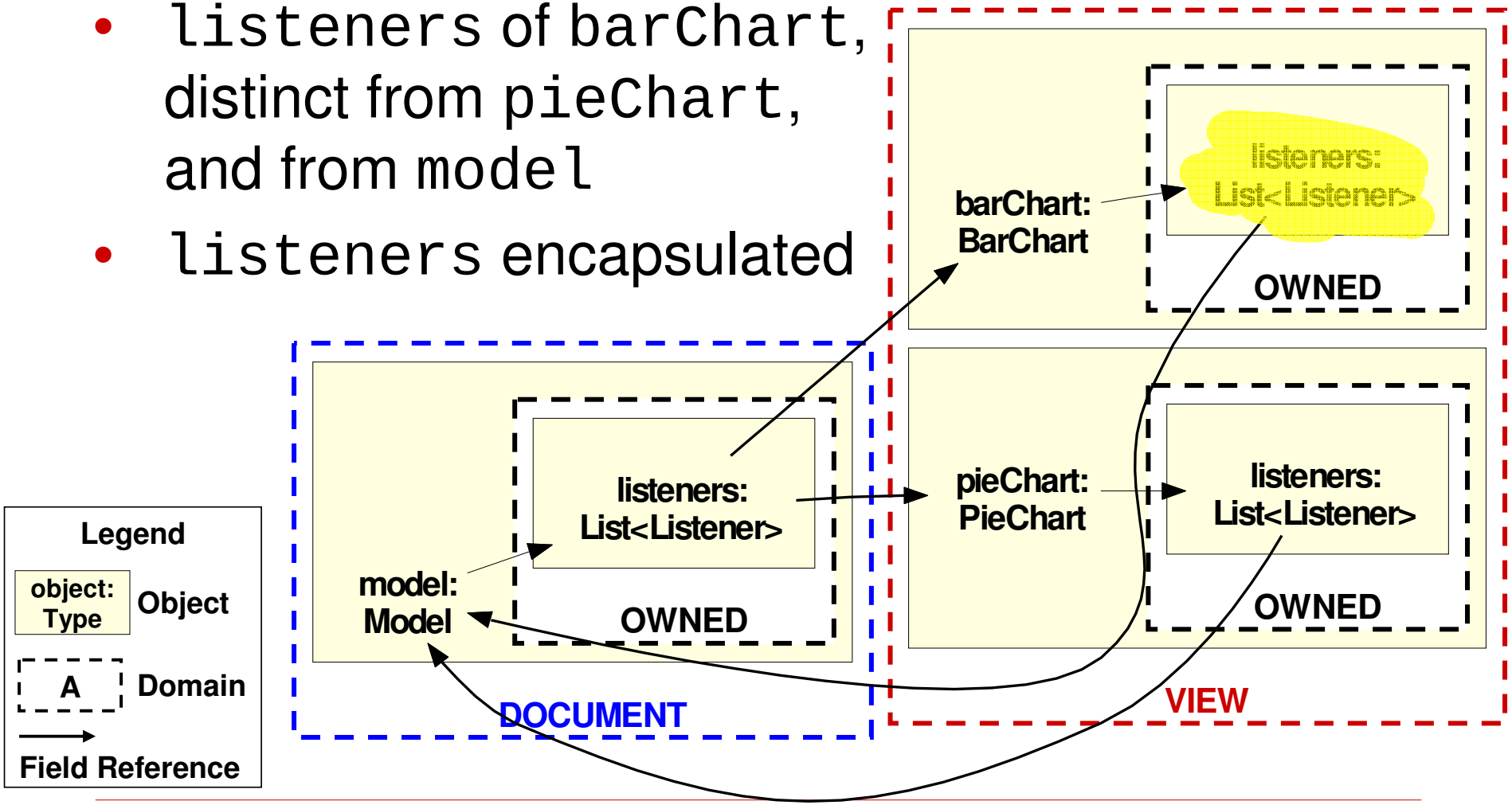
Object graph complements information in class diagram

- Show top-level **objects**
- (+) indicates hidden sub-structure
- Data structures part of other objects



Hierarchy enables expanding level of detail and varying level of abstraction

- `listeners` of `barChart`, distinct from `pieChart`, and from `model`
- `listeners` encapsulated

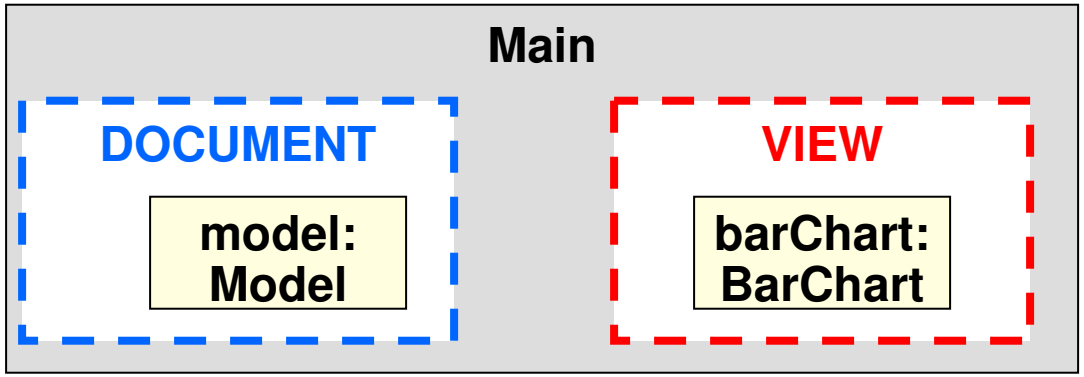


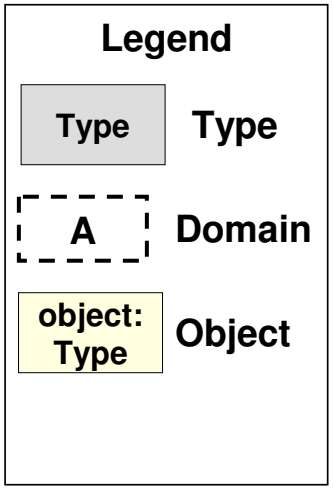
Analysis

Approach combines: (1) type annotations and (2) static analysis

1. Add ownership domain annotations
   [Aldrich and Chambers, ECOOP'04]
   - Typecheck and address warnings
   - Possibly use ownership inference
     [Ma and Foster, OOPSLA'07] [Milanova, IWACO'08] [Aldrich et al., OOPSLA'02]
2. Run static analysis
   - Extract object graph
   - Refine annotations as needed

Group objects into *ownership domains*



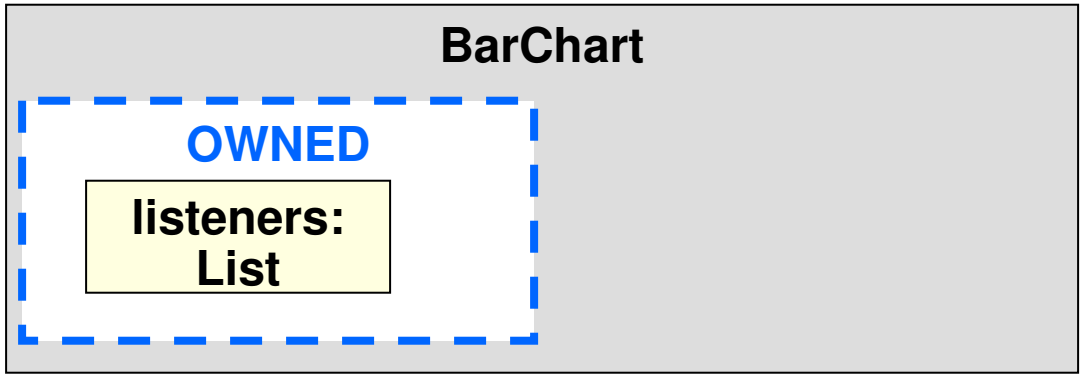


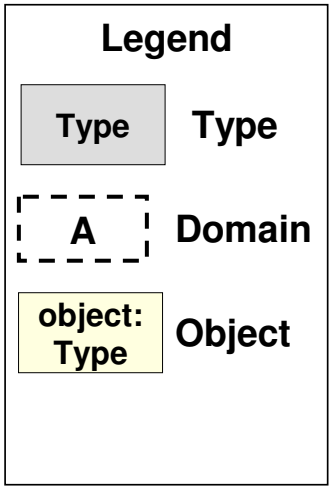

```
class Main {
  domain DOCUMENT, VIEW;
  DOCUMENT Model model;
  VIEW BarChart barChart;
  ...
}
```

Declarations are simplified

- Each object **in exactly one domain**
- No ownership transfer; no multiple ownership

Domains can be declared inside each class

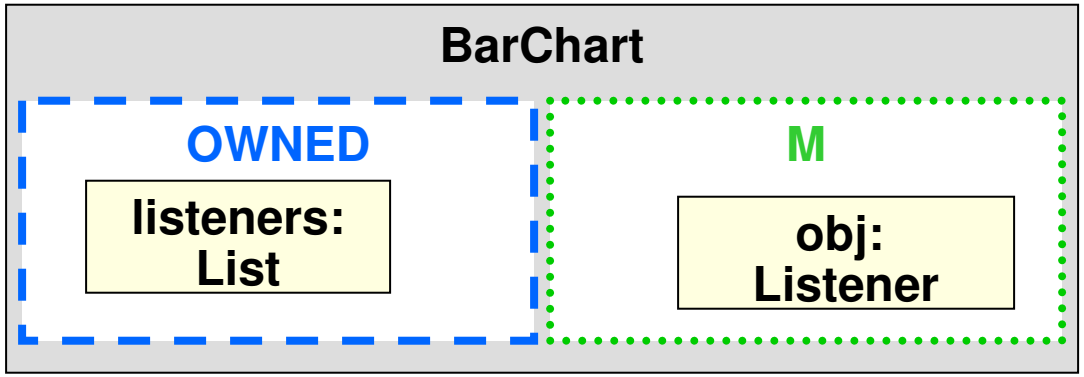





```
class BarChart {
  domain OWNED;
  OWNED List listeners;
  ...
}
```

Declarations are simplified

Domain parameters allow sharing objects




```
class BarChart < M > {
  domain OWNED;
  OWNED List< M Listener> listeners;
}
class Main {
  domain DOCUMENT, VIEW;
  VIEW BarChart<DOCUMENT> barChart;
  ...
}
```

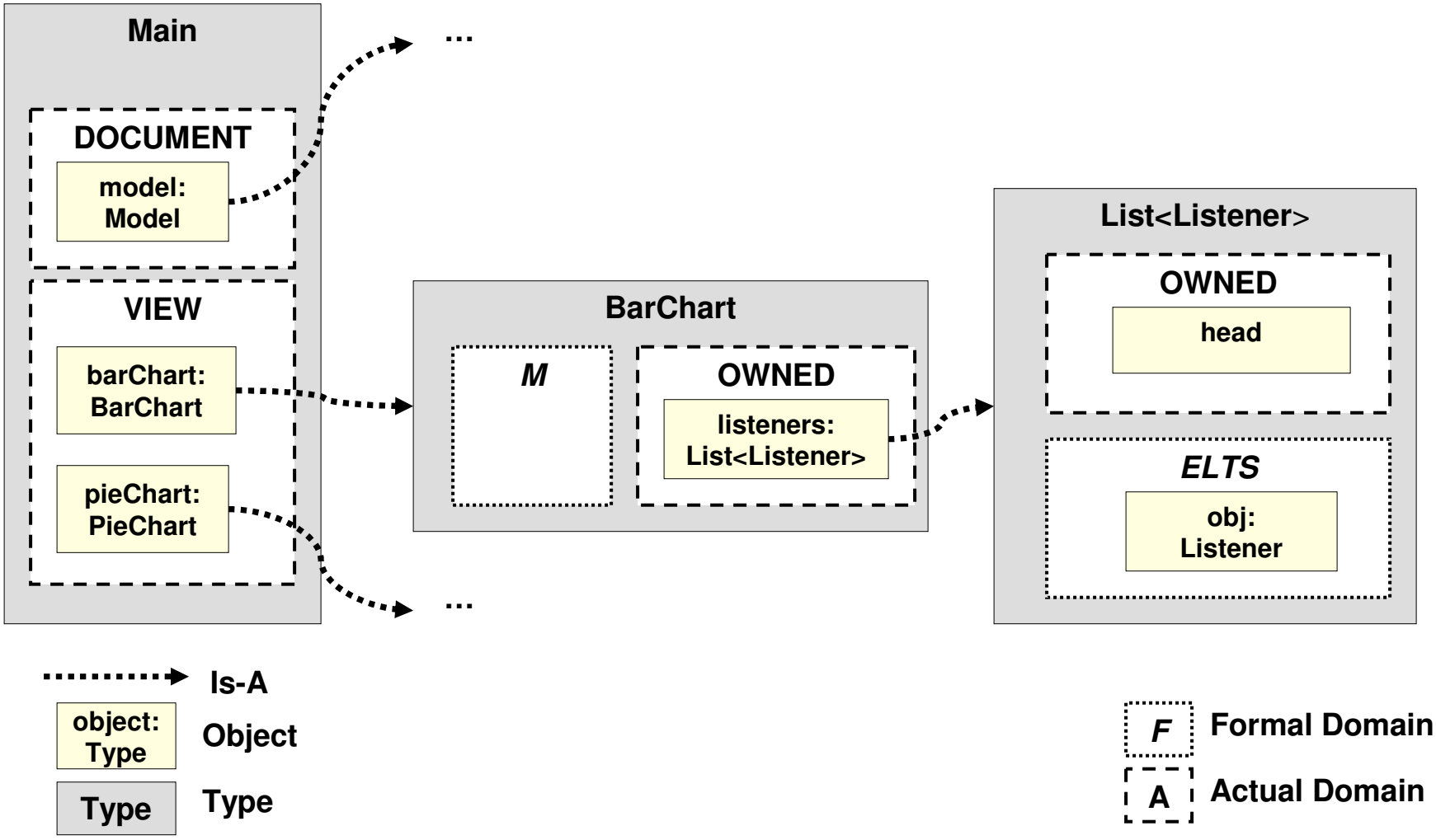
Bind FormalDomain → ActualDomain

BarChart :: M → Main :: DOCUMENT

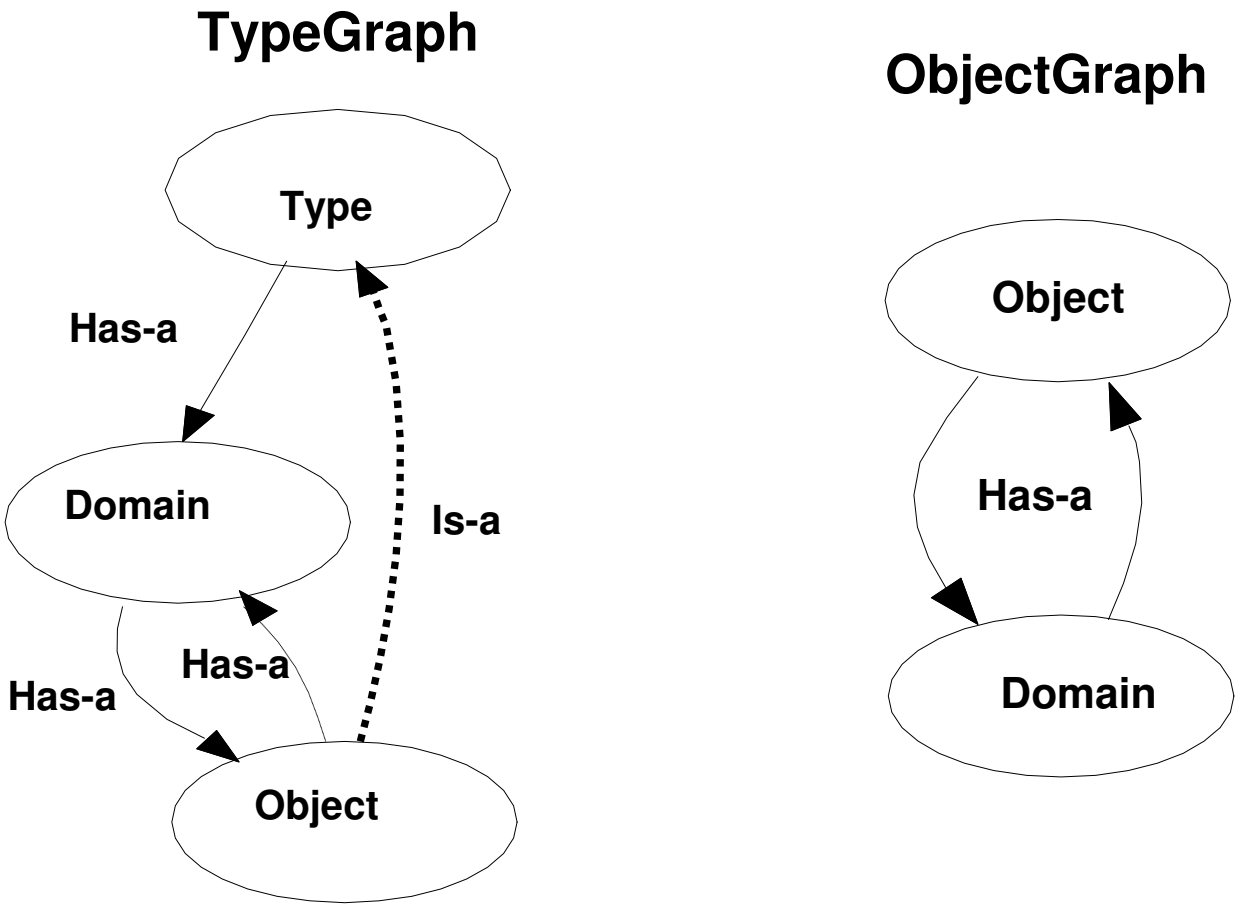
Static analysis to extract object graph

- Build **TypeGraph**
  - Visitor over program's Abstract Syntax Tree
  - Represents type structure of objects in code
- Convert **TypeGraph** to **ObjectGraph**
  - **Instantiates the types** in the TypeGraph
  - Shows only objects and domains

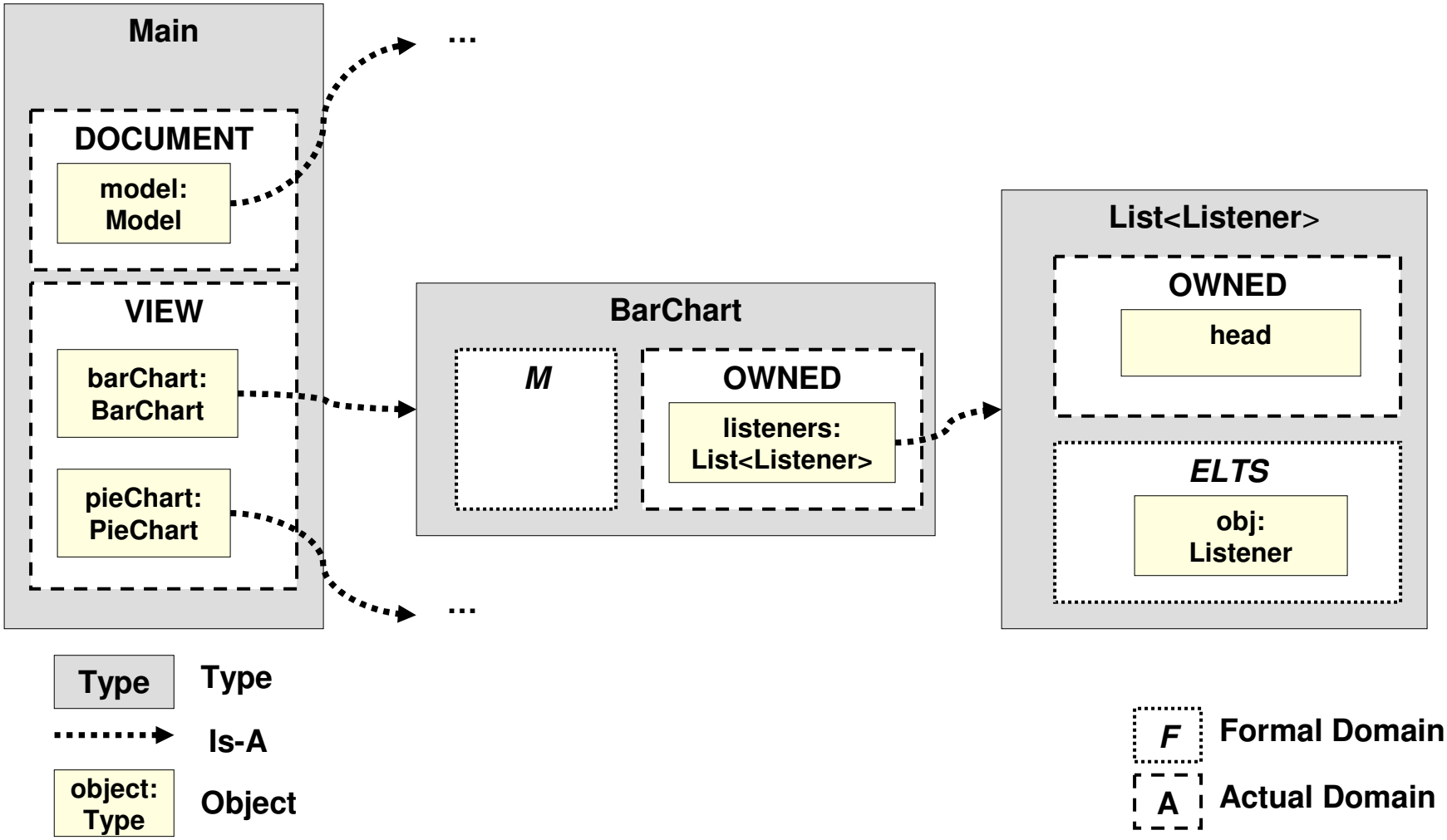
TypeGraph: show types, domains inside types, and objects in domains



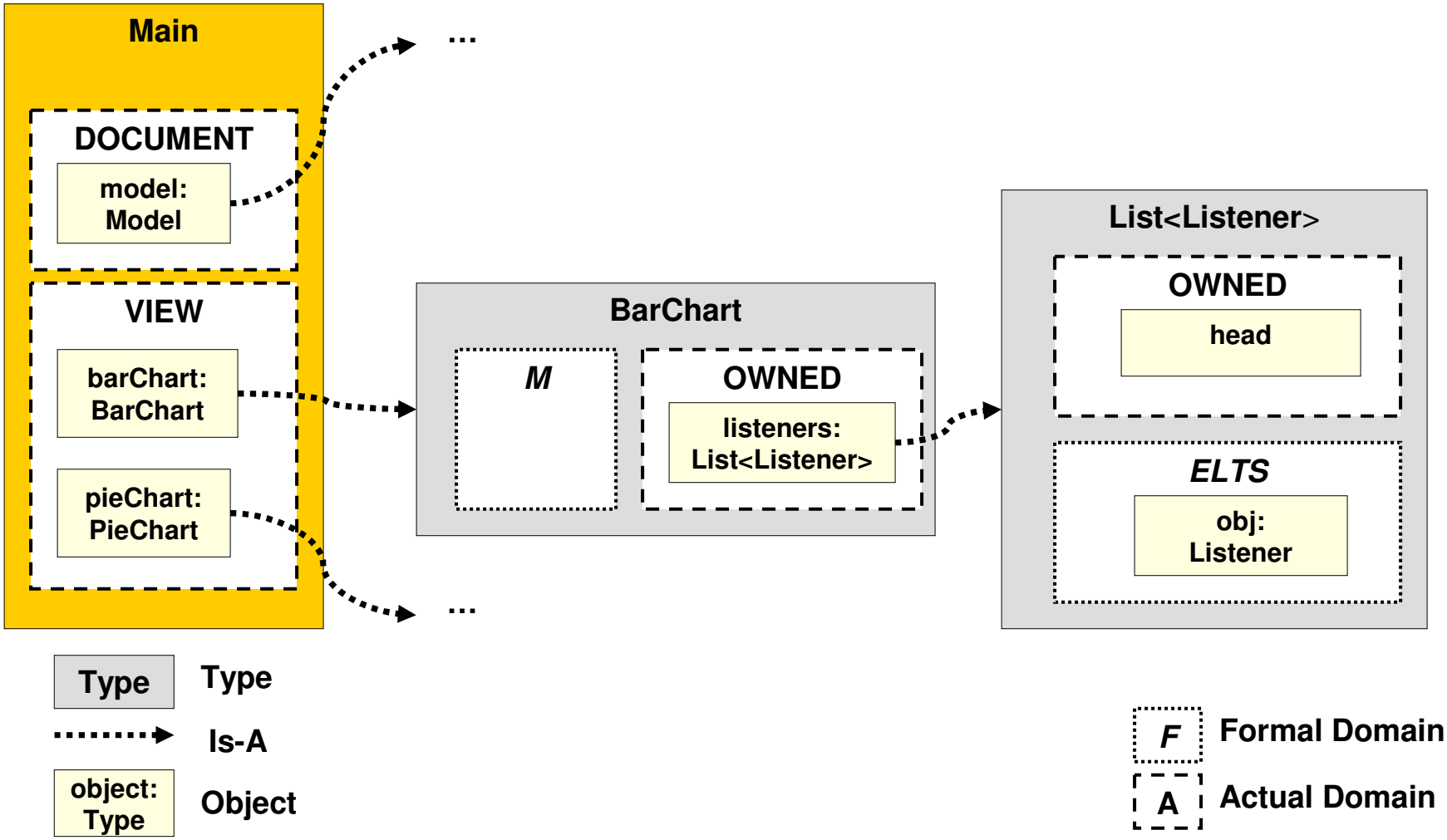
Challenge: TypeGraph does not show children objects of a given object



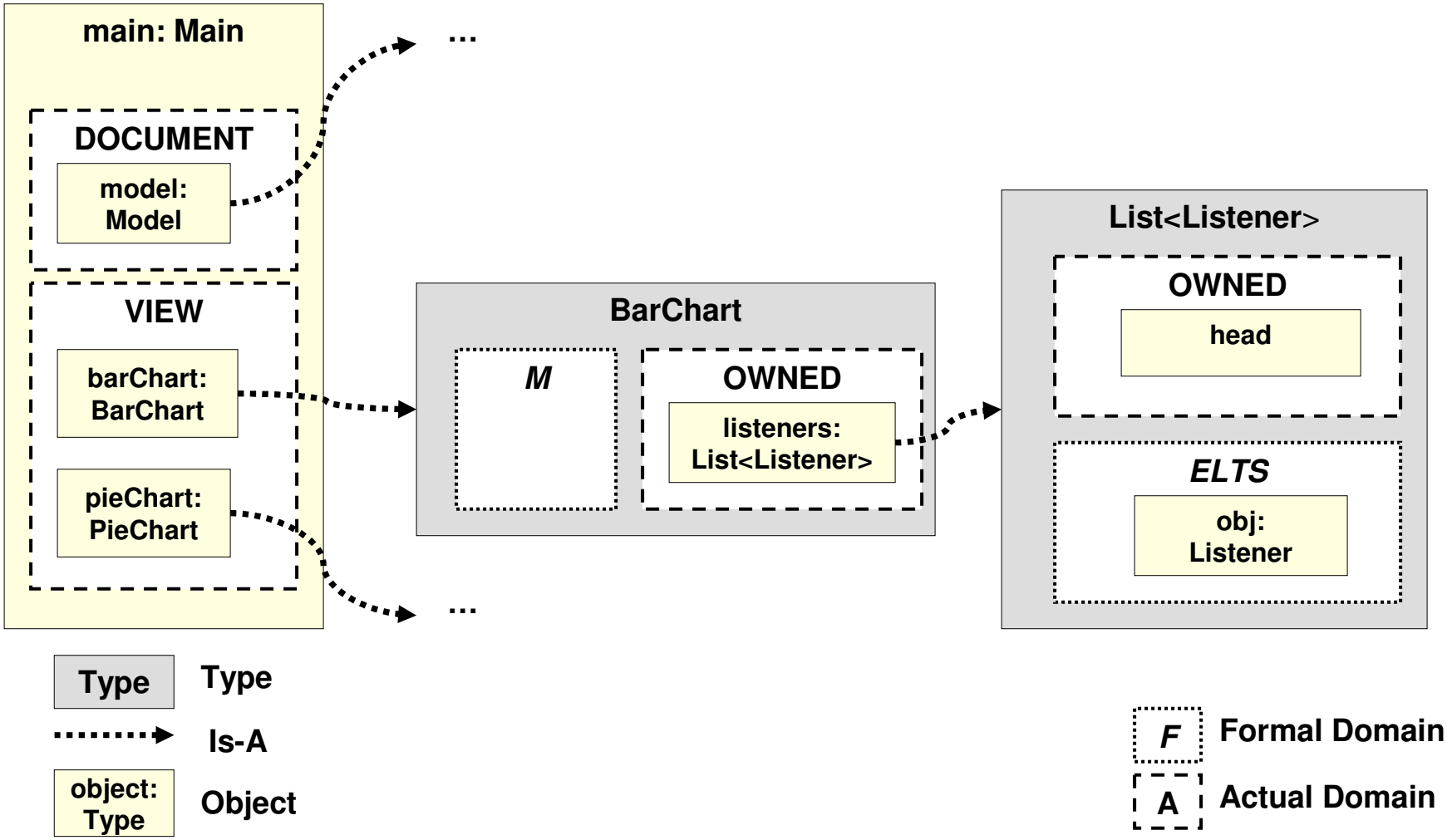
ObjectGraph: instantiate types, starting with root (user selected)



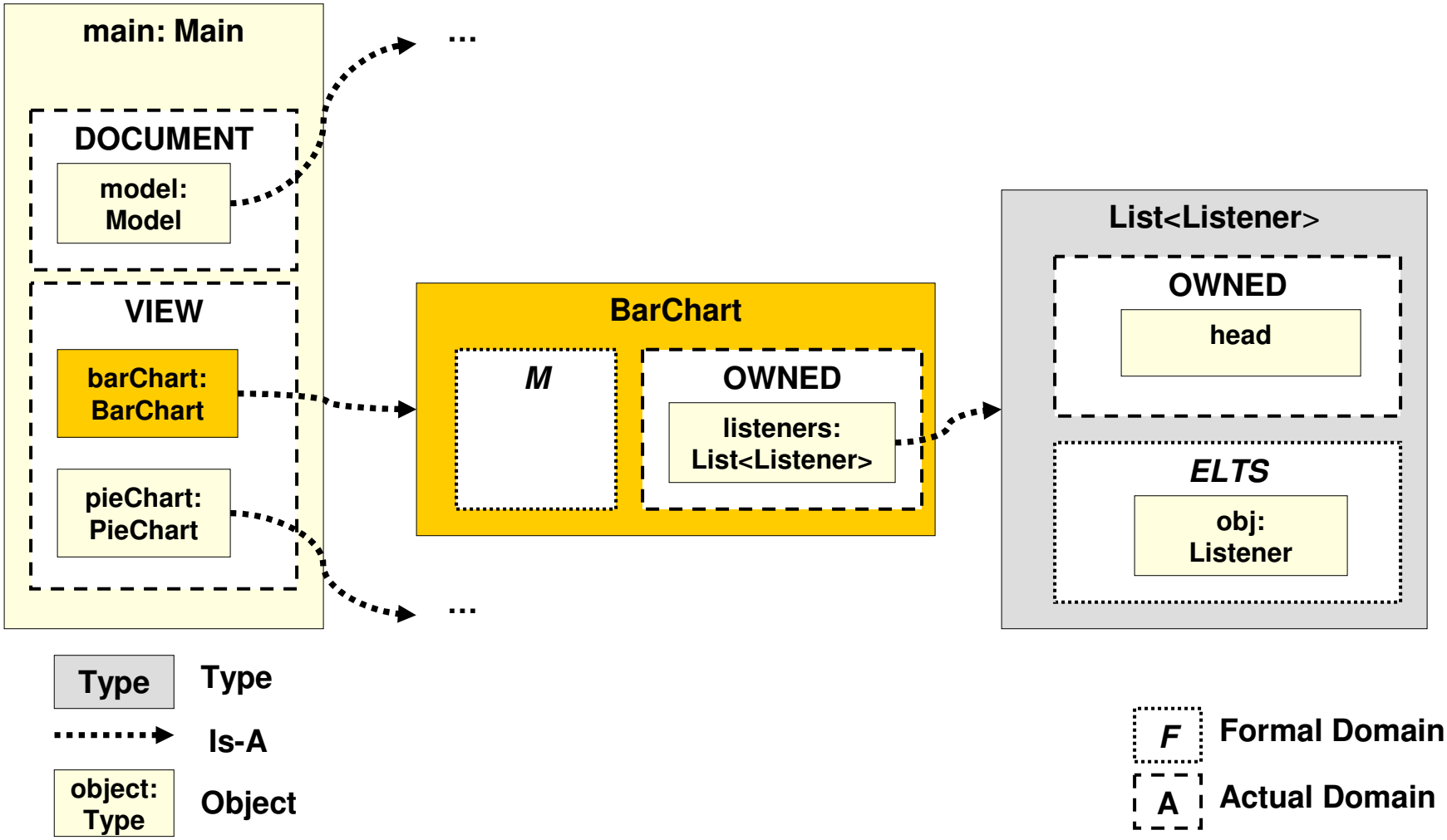
ObjectGraph: instantiate types, starting with root



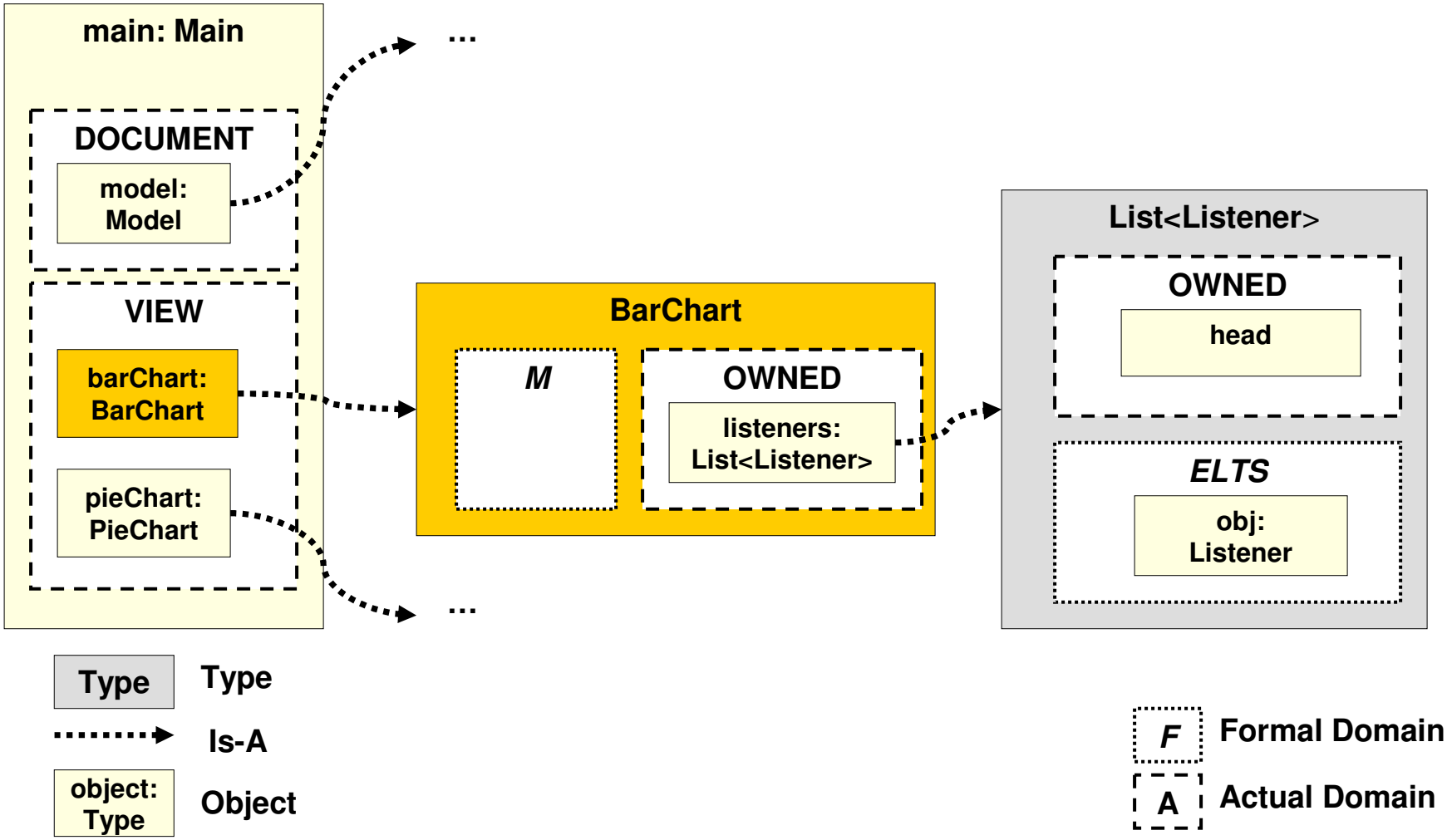
ObjectGraph: instantiate types, starting with root



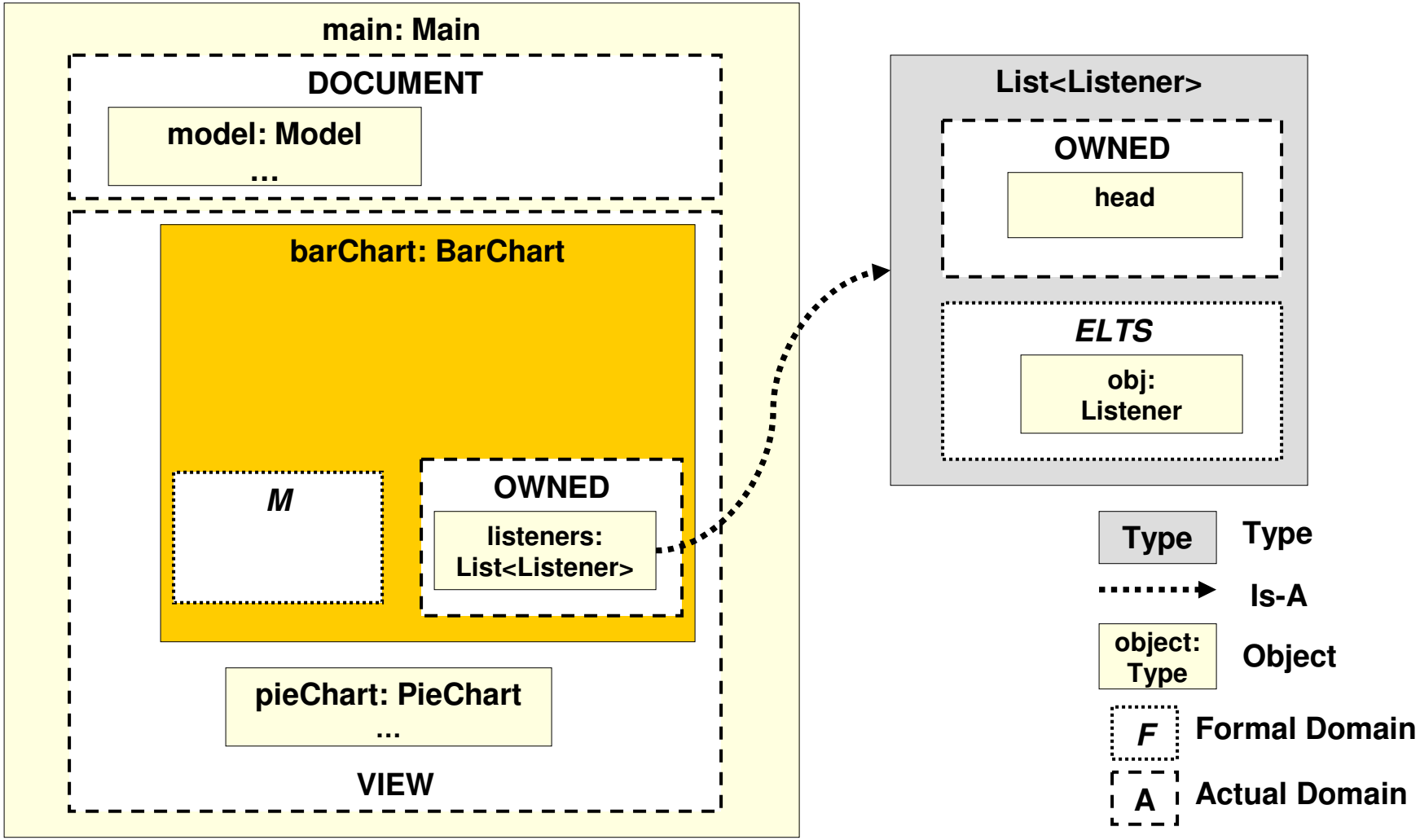
ObjectGraph: instantiate types, show domains and objects inside domains



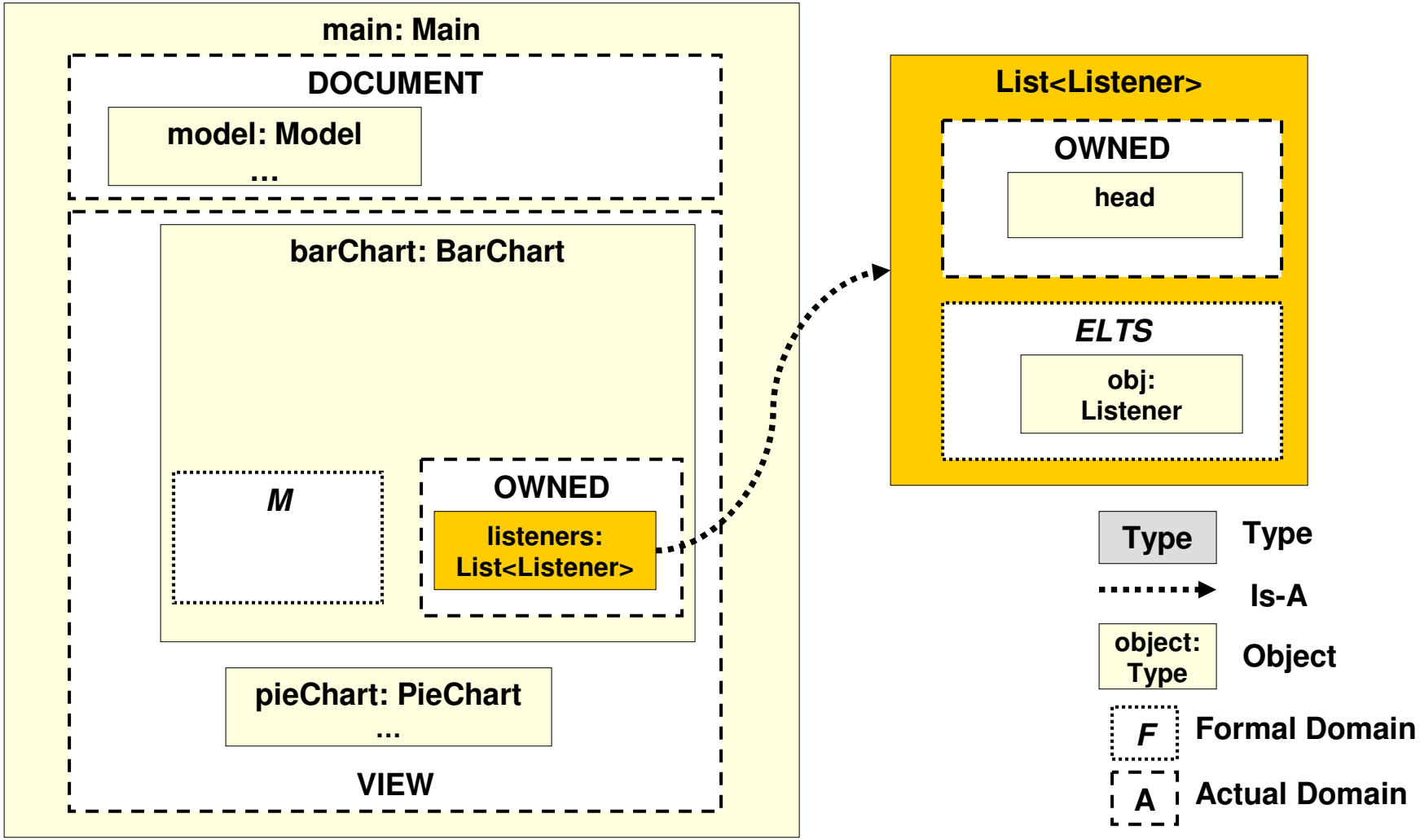
ObjectGraph: instantiate types, show domains and objects inside domains



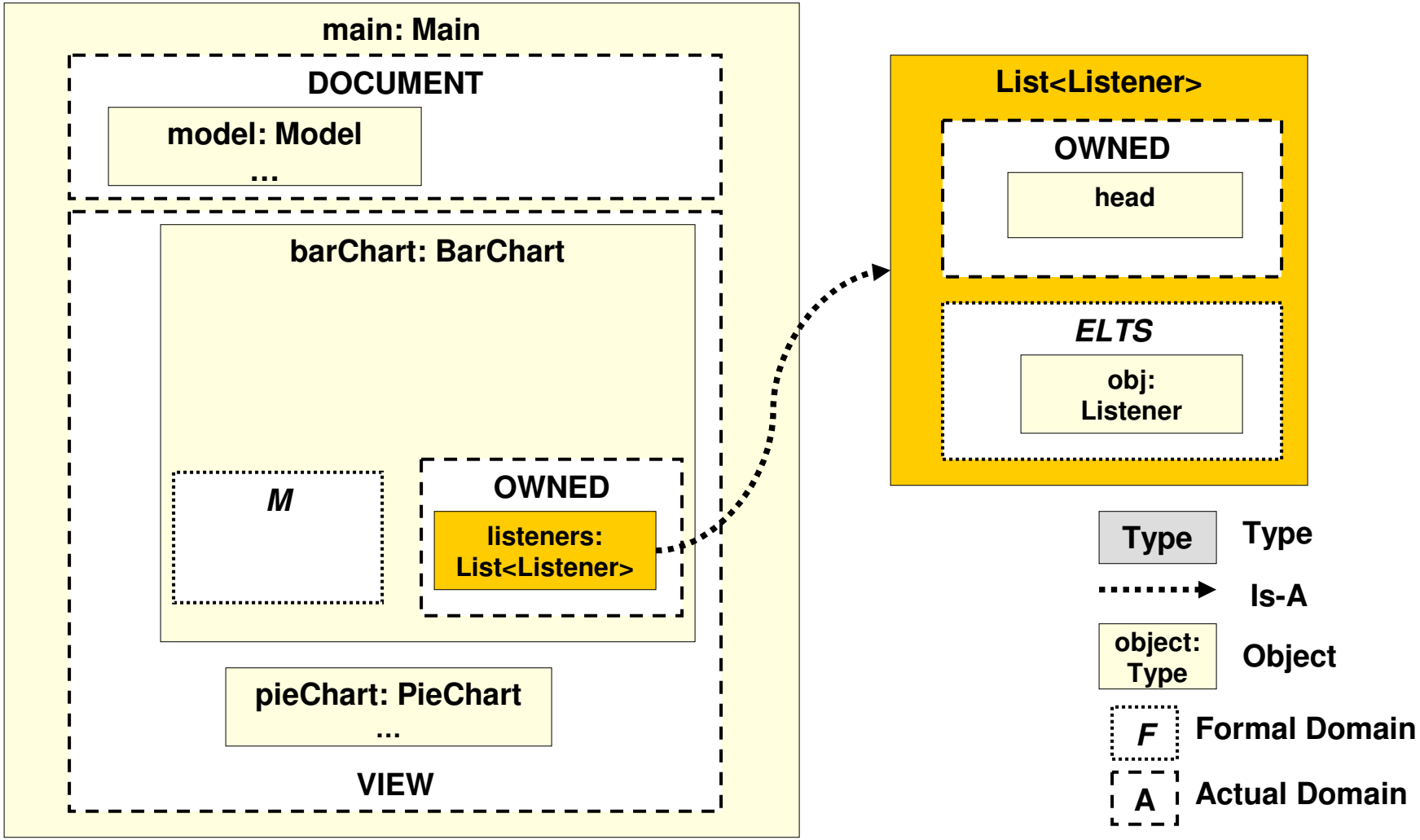
ObjectGraph: instantiate types, show domains and objects inside domains



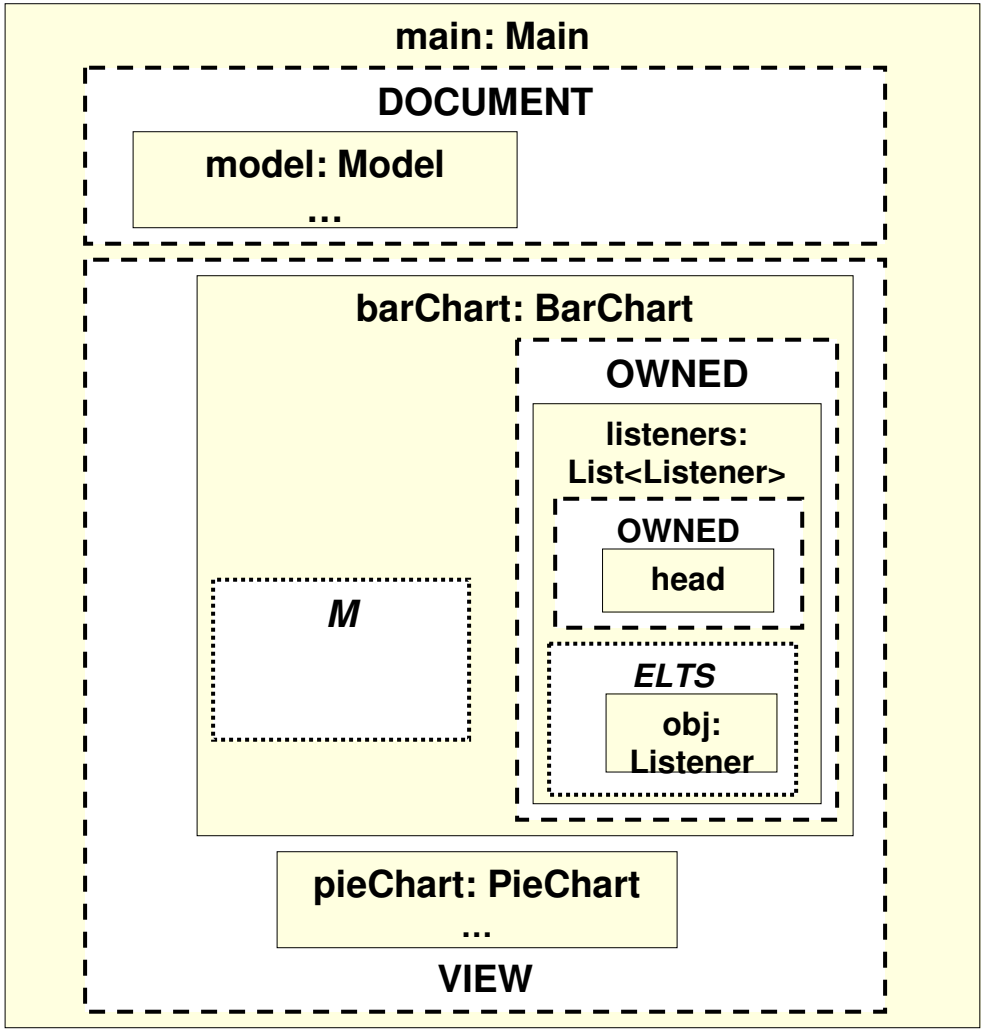
ObjectGraph: instantiate types, show domains and objects inside domains

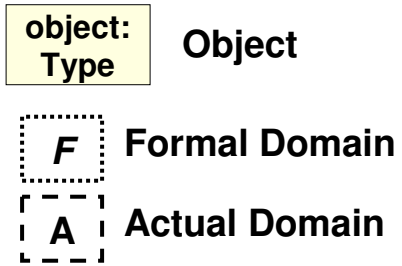


ObjectGraph: instantiate types, show domains and objects inside domains



ObjectGraph: instantiate types, show domains and objects inside domains



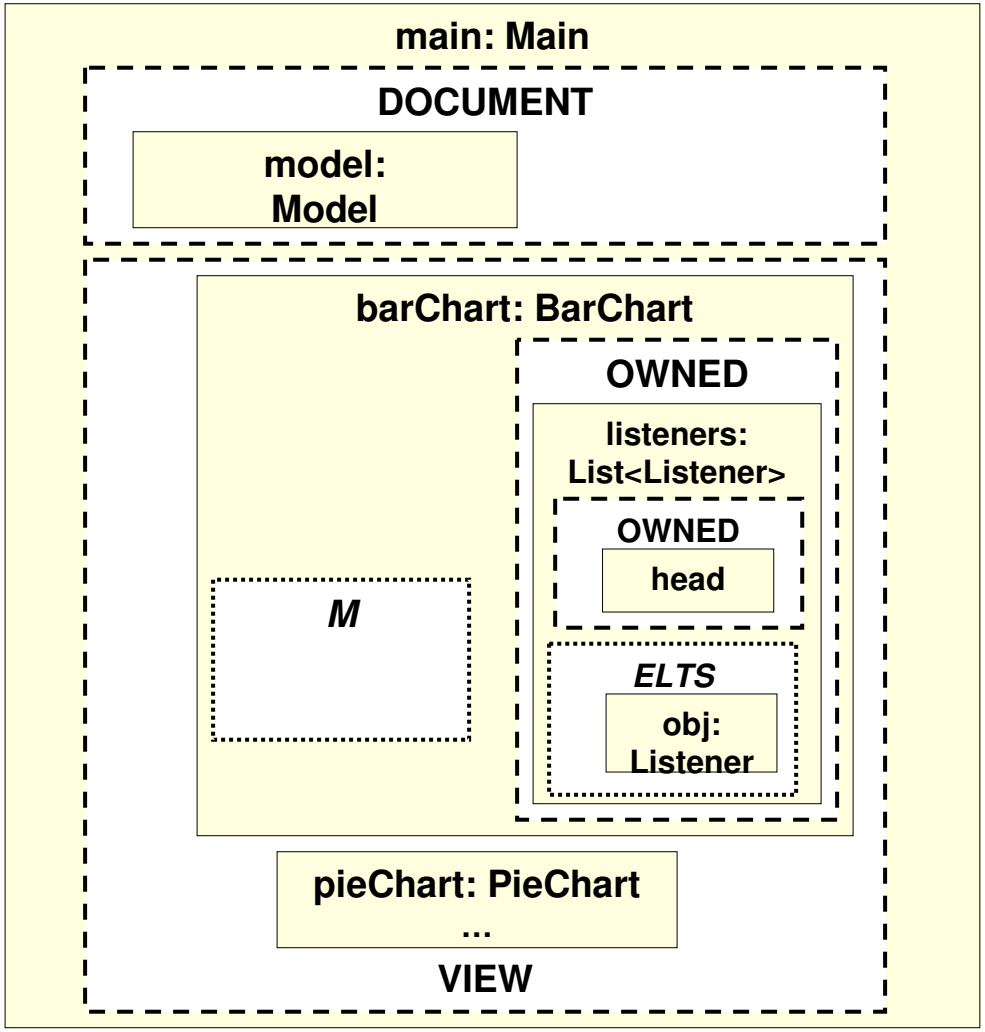


Challenge: unbounded number of objects, based on different executions

- *Invariant: Summarize multiple objects in a domain with one canonical object*

- *Invariant: Merge two objects of the "same type" that are in the same domain*
  - I.e., same declared type, or subtype thereof
  - Or of compatible types (more later)

ObjectGraph: merge equivalent objects inside a given domain



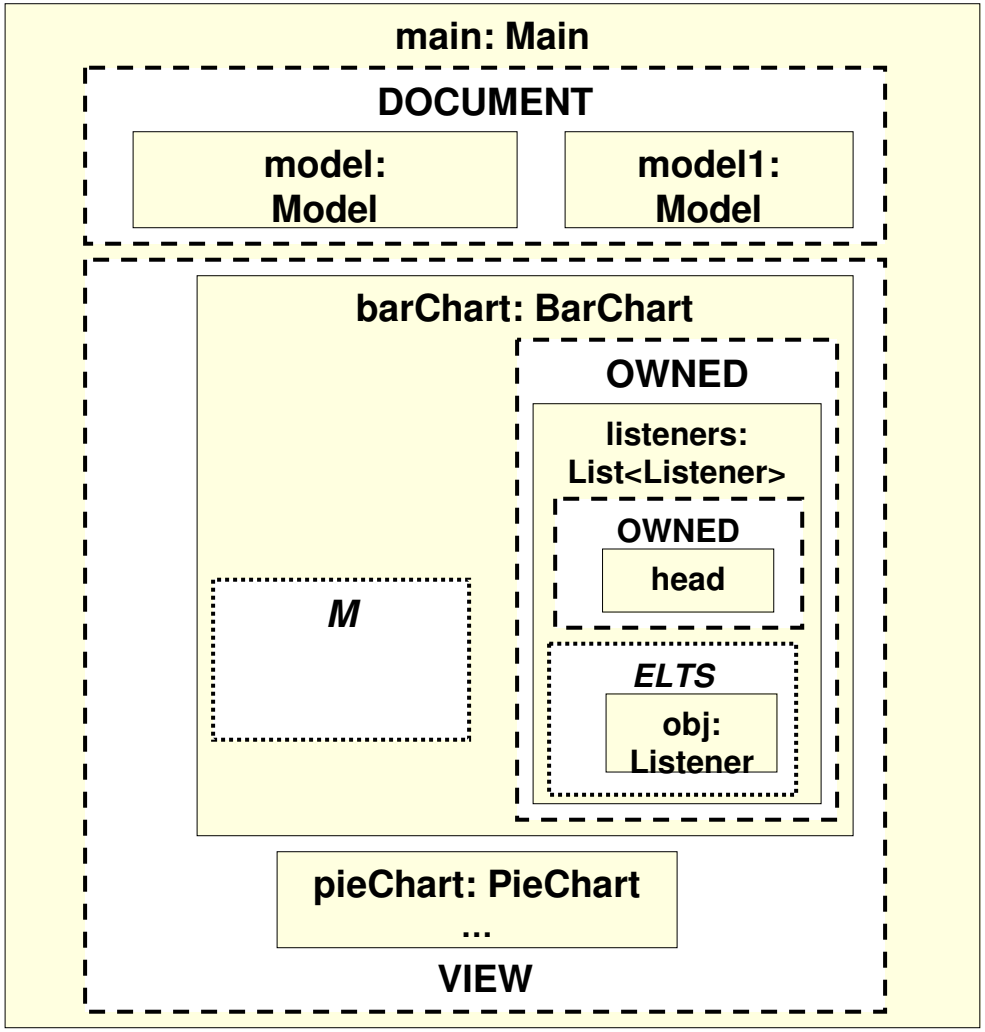

```
class Main {
  domain DOCUMENT, VIEW;

  DOCUMENT Model model;


  ...
}
```

object: Type — Object

F — Formal Domain

A — Actual Domain

ObjectGraph: merge equivalent objects inside a given domain



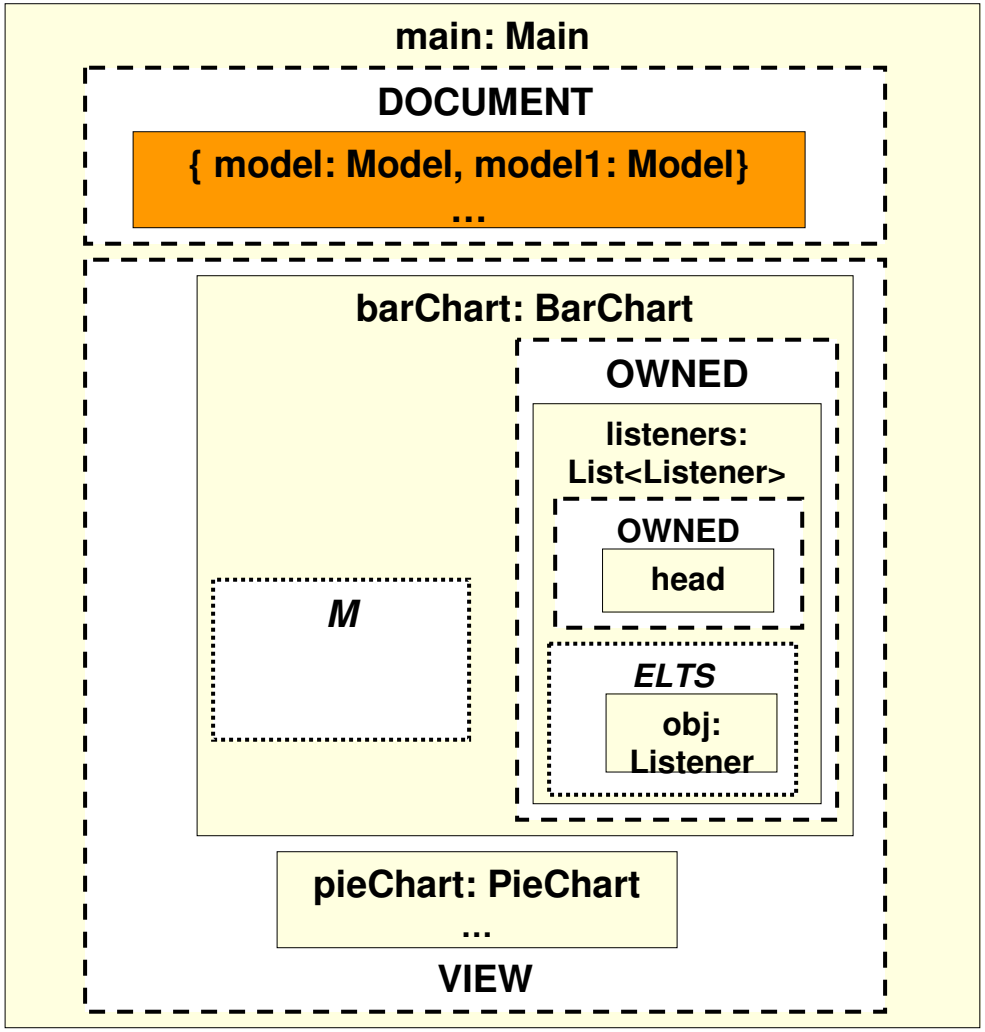

```
class Main {
  domain DOCUMENT, VIEW;

  DOCUMENT Model model;
  DOCUMENT Model model1;

  ...
}
```

object: Type — Object

F — Formal Domain

A — Actual Domain

ObjectGraph: merge equivalent objects inside a given domain




```
class Main {
  domain DOCUMENT, VIEW;

  DOCUMENT Model model;
  DOCUMENT Model model1;

  ...
}
```

object: Type — Object

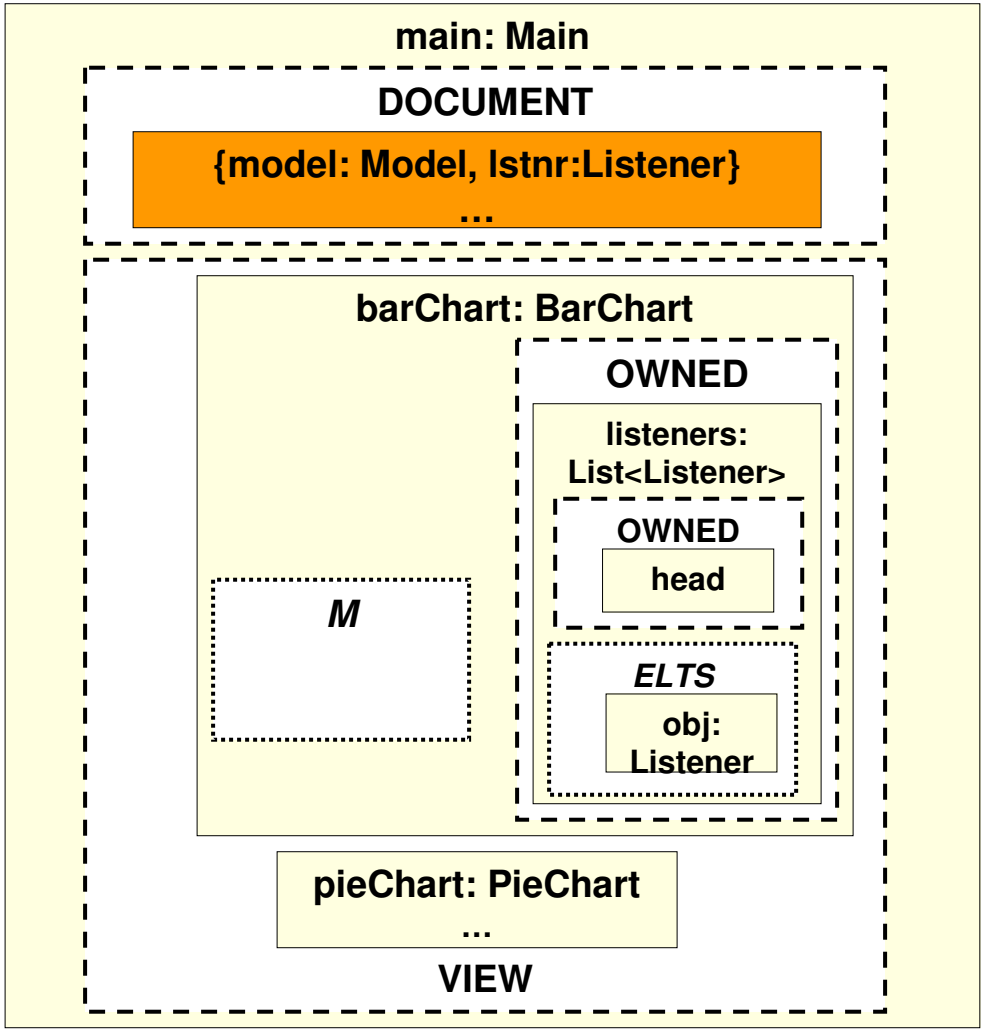
F — Formal Domain

A — Actual Domain

Challenge: TypeGraph does not reflect possible aliasing

- *Invariant: the same object should not appear multiple times in the **ObjectGraph***
- Ownership domain annotations give some precision about aliasing:
  - Two objects in different domains cannot alias
  - Two objects in same domain *may* alias

ObjectGraph: merge objects, in one domain, that *may* alias, based on types




```
class Main {
  domain DOCUMENT, VIEW;

  DOCUMENT Model model;
  DOCUMENT Listener lstnr;

  ...
}

class Model implements Listener {
  ...
}
```

object: Type — **Object**

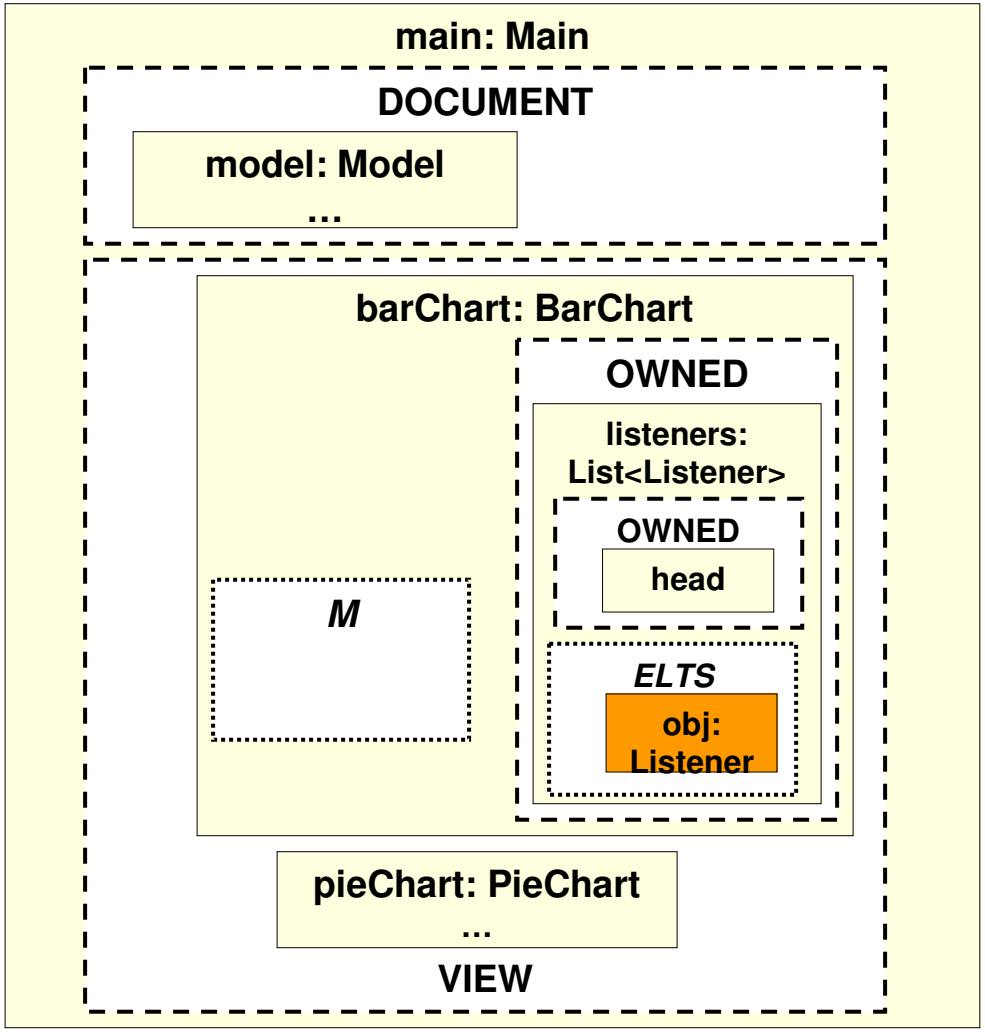
F — **Formal Domain**

A — **Actual Domain**

Challenge: TypeGraph does not show all objects in each domain

- Object can be declared in domain parameter
- Domain parameter bound to other domain
- *Invariant: In the ObjectGraph, each object that is in a given domain must appear where that domain is declared*
- **Pull each object** declared inside formal domain parameter into each domain bound to the formal domain parameter

ObjectGraph: pull objects from formal domains to actual domains



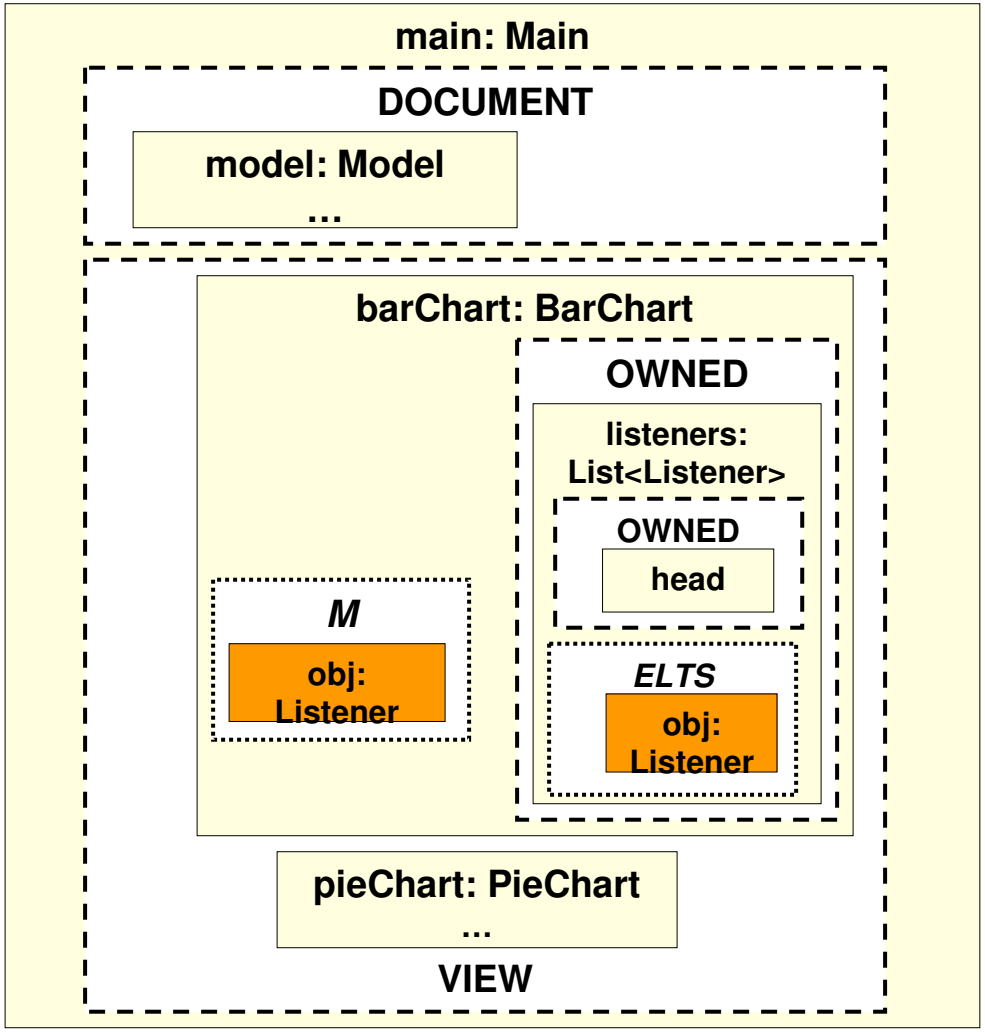

```
class BarChart <M> {
 domain OWNED;

 OWNED List<M Listener> listeners;
}

class List<ELTS T> {
 // ELTS is for List elements
 // T is generic type parameter
 // Virtual field
 ELTS T obj;
}
```

object: Type — Object

F — Formal Domain

A — Actual Domain

ObjectGraph: pull objects from formal domains to actual domains



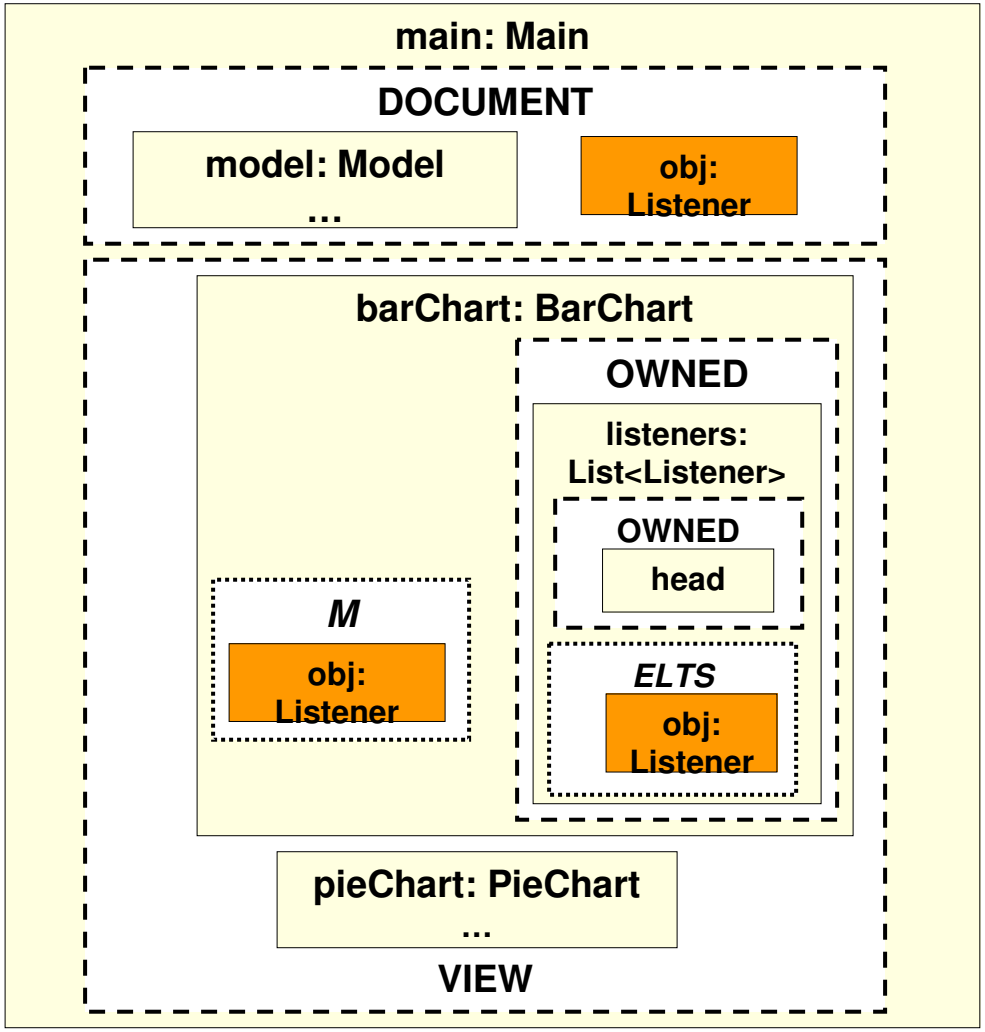

```
class BarChart <M> {
 domain OWNED;
 // List::ELTS    BarChart::M
 OWNED List<M Listener> listeners;
}

class List<ELTS T> {
 // ELTS is for List elements
 // T is generic type parameter
 // Virtual field
 ELTS T obj;
}
```

object: Type — Object

F — Formal Domain

A — Actual Domain

ObjectGraph: pull objects from formal domains to actual domains



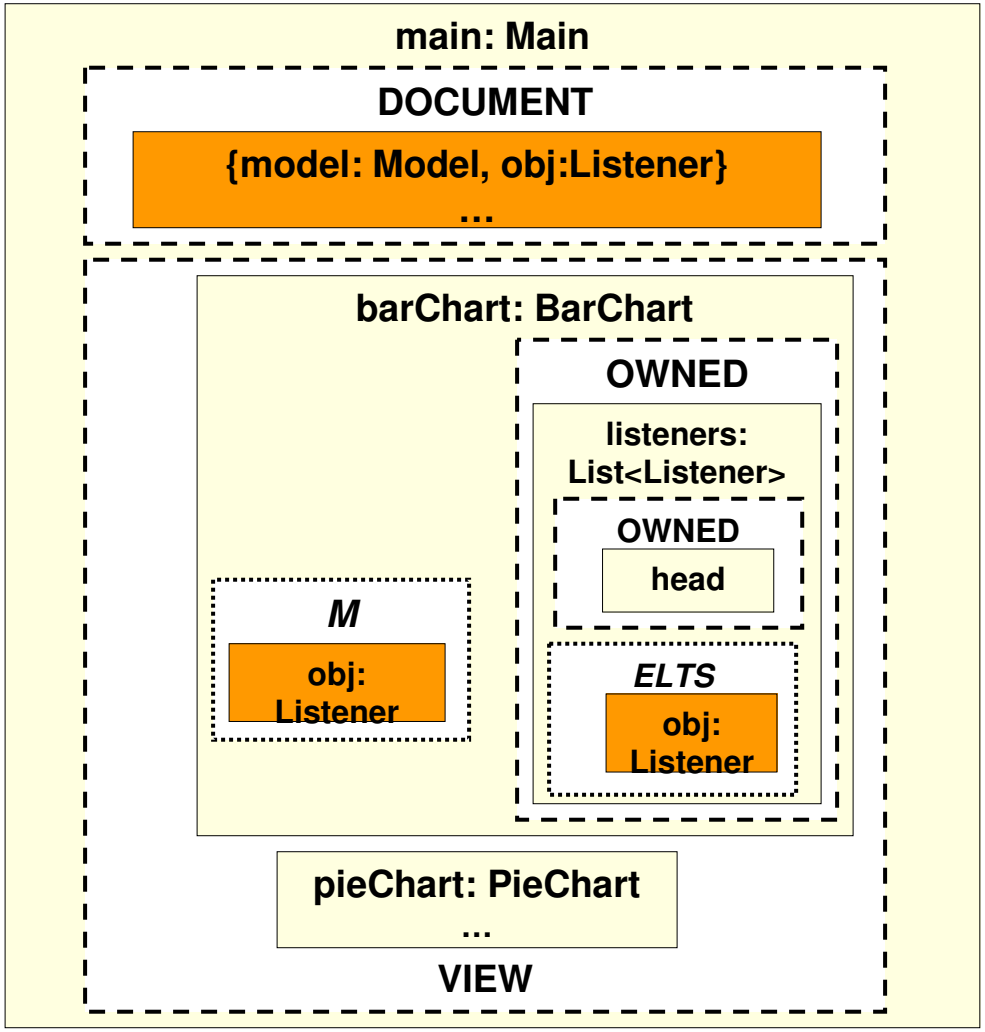

```
class Main {
 domain DOCUMENT, VIEW;
 // BarChart::M    Main::DOCUMENT
 VIEW BarChart<DOCUMENT> barChart;
   ...
}

class BarChart<M> {
 domain OWNED;
 OWNED List<M Listener> listeners;
}
```

object: Type — **Object**

F — **Formal Domain**

A — **Actual Domain**

ObjectGraph: again, merge objects, in one domain, that *may* alias



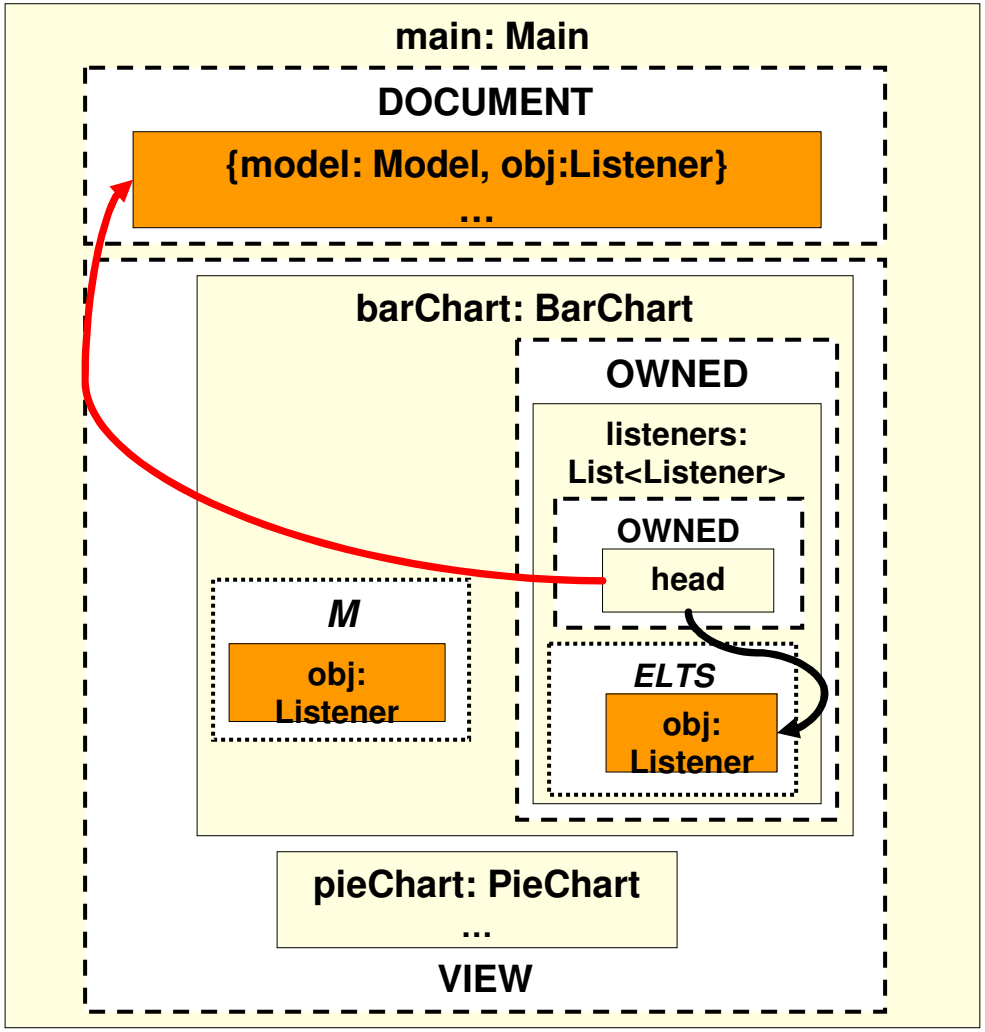

```
class Model implements Listener {
  ...
}
```

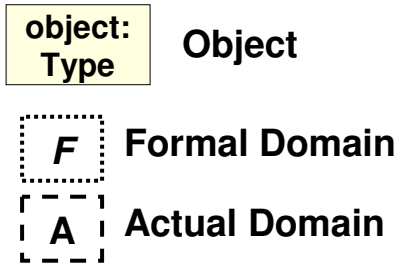
object: Type — Object

F — Formal Domain

A — Actual Domain

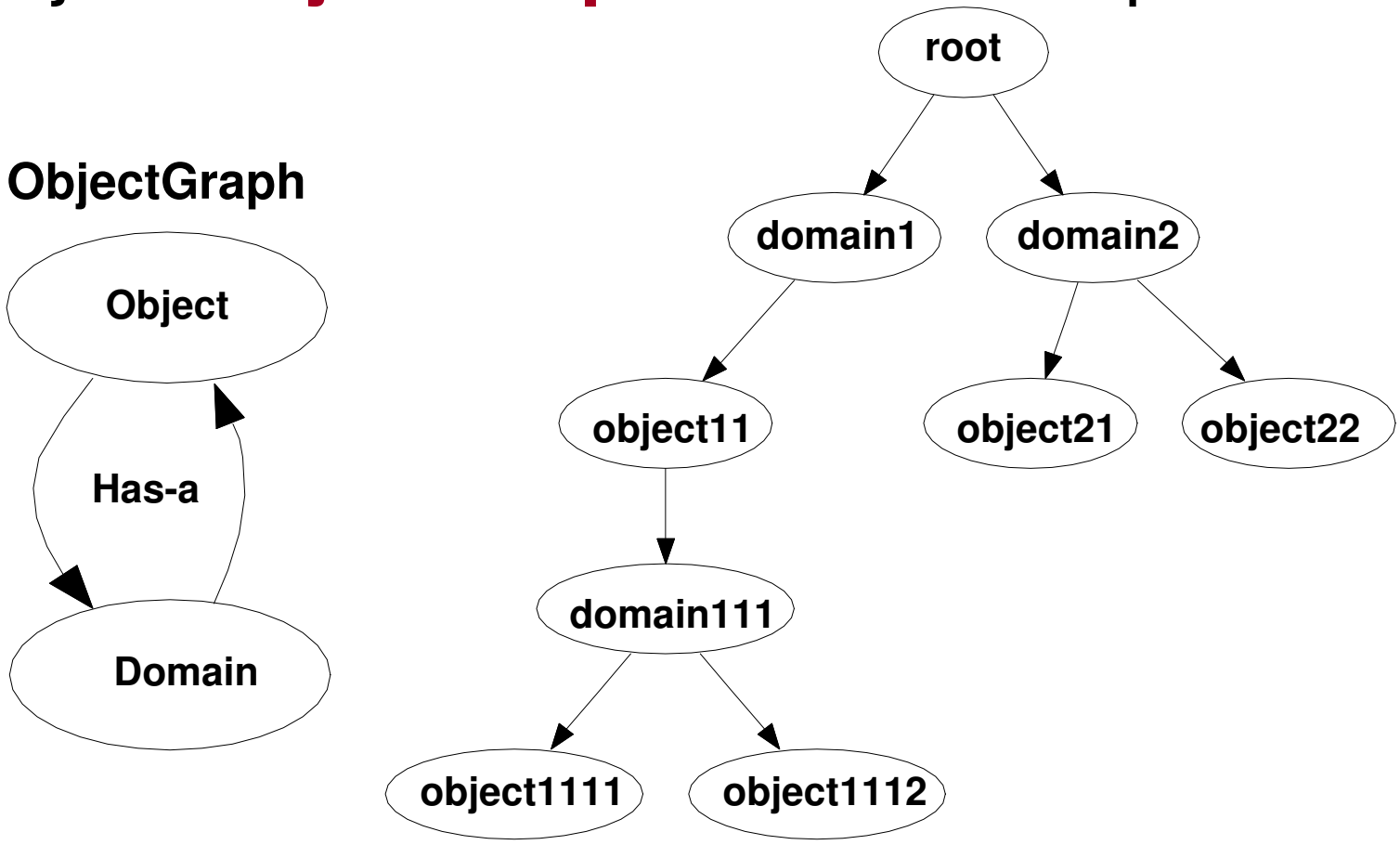
ObjectGraph: add edges to represent points-to relations, incl. to pulled objects





Challenge: ObjectGraph can have cycles

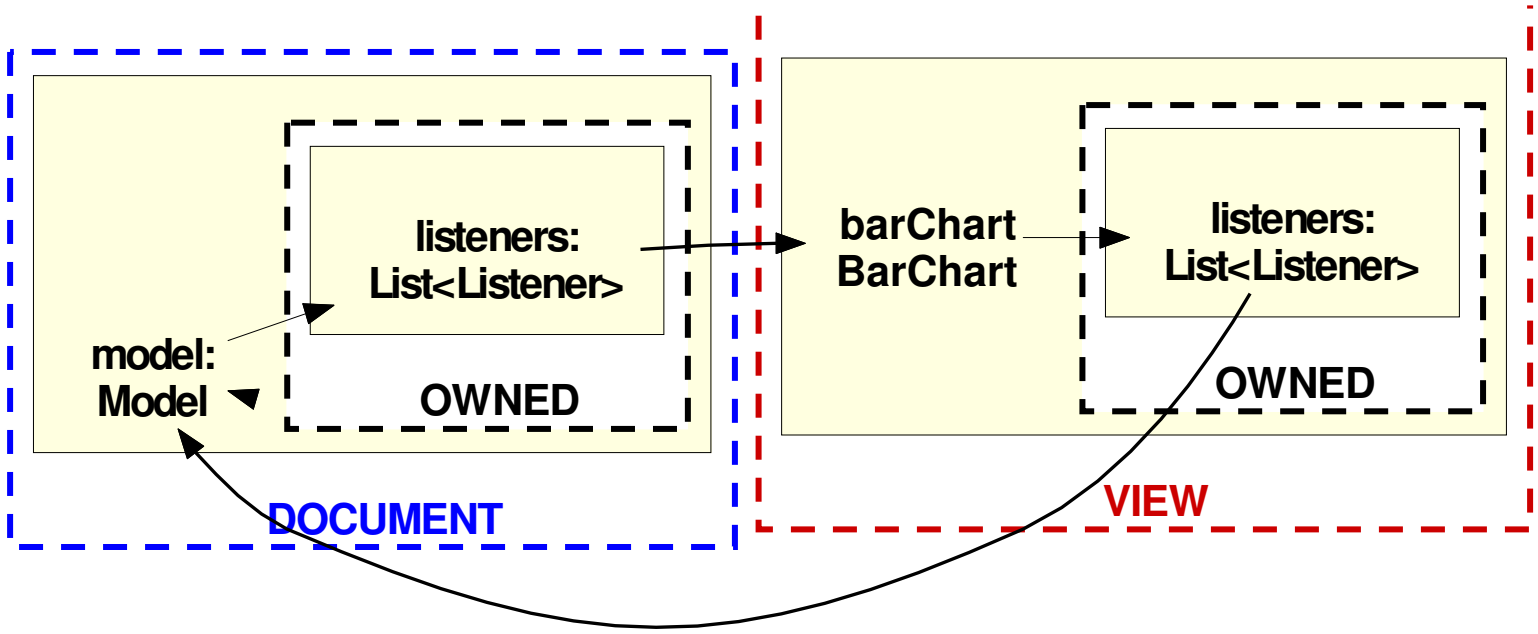
- Project **ObjectGraph** to limited depth



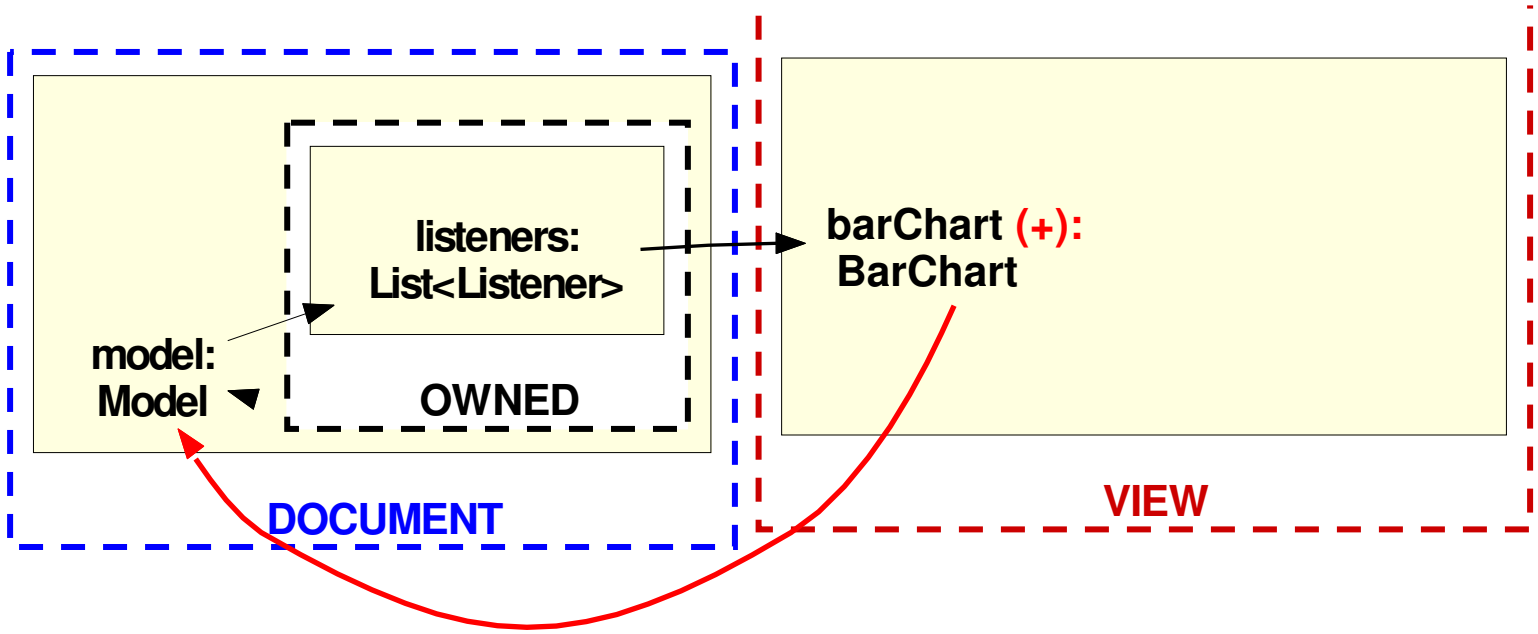
Challenge: edges could be due to sub-objects that point to external objects

- *Invariant: show all object relations, even ones due to objects in sub-structures*
- Lift **edge** to parent object when hidden sub-object points to external objects

Example of edge lifting



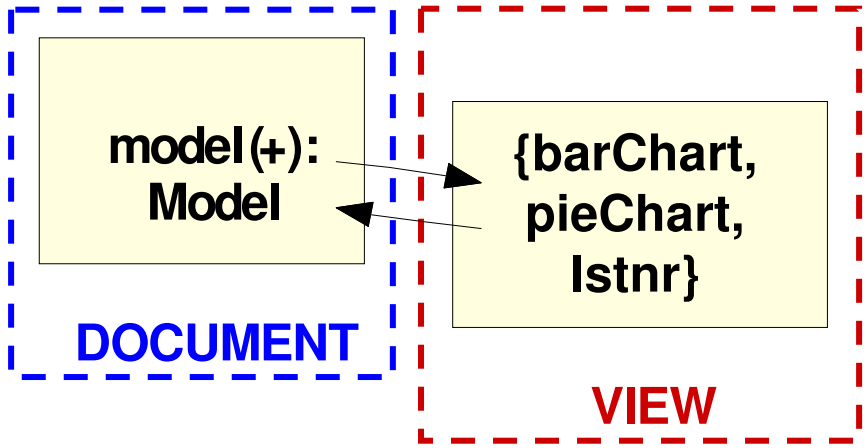
Example of edge lifting



Abstraction by types

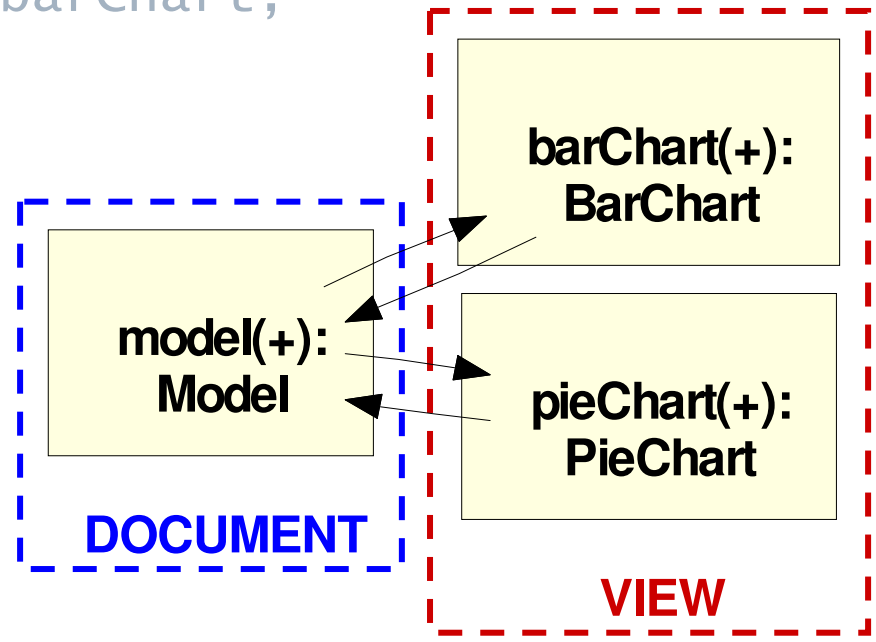
Challenge: using declared types could lead to excessive merging (imprecision)

```
class Main {
  domain DOCUMENT, VIEW;
  DOCUMENT Model model;
  VIEW BarChart barChart;
  VIEW PieChart pieChart;
  VIEW Listener lstnr = barChart;
  ...
}
```



TypeGraph: consider only object allocations (Instantiation-Based View)

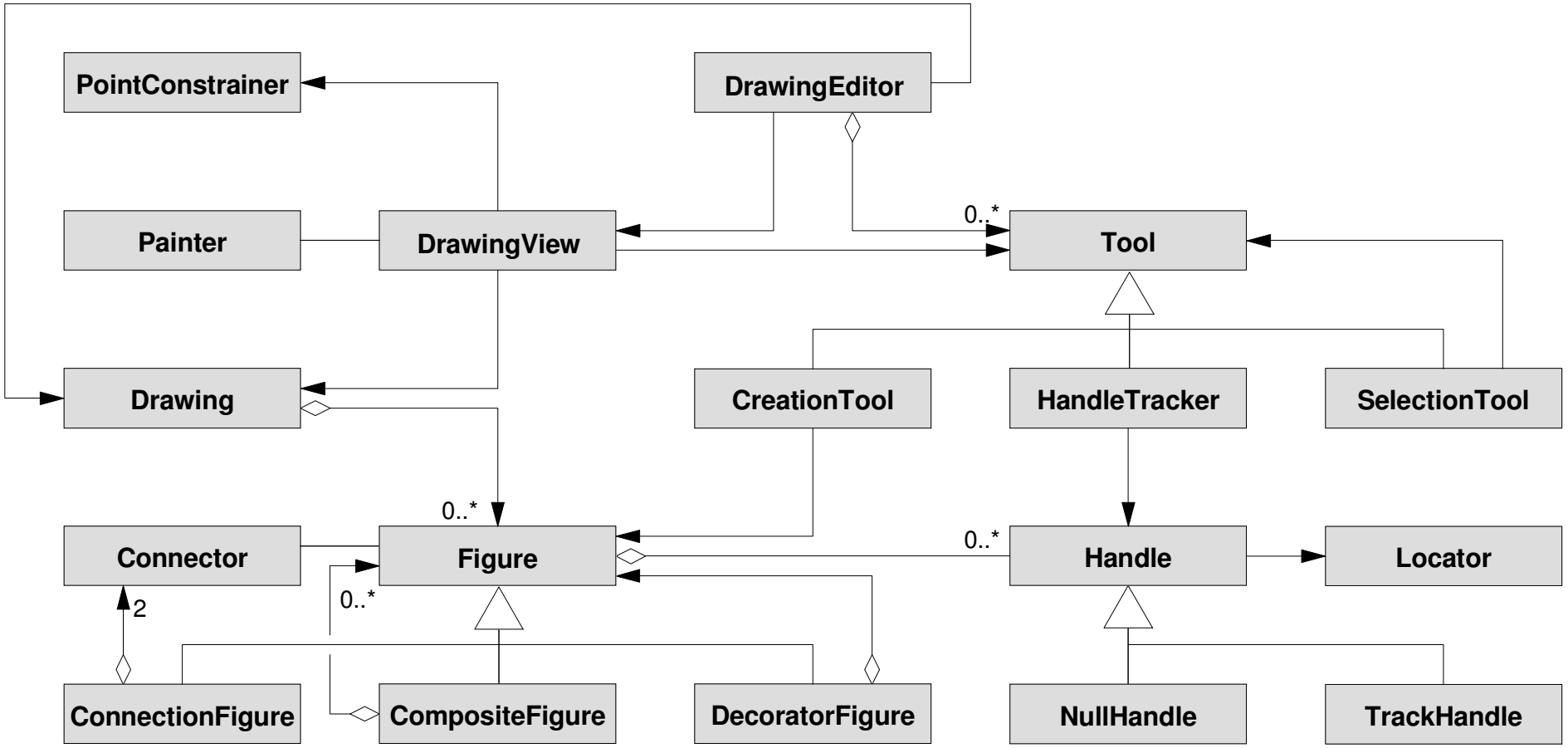
```
class Main {
  domain DOCUMENT, VIEW;
  DOCUMENT Model model = new Model();
  VIEW BarChart barChart = new BarChart();
  VIEW PieChart pieChart = new PieChart();
  VIEW Listener lstnr = barChart;
  ...
}
```



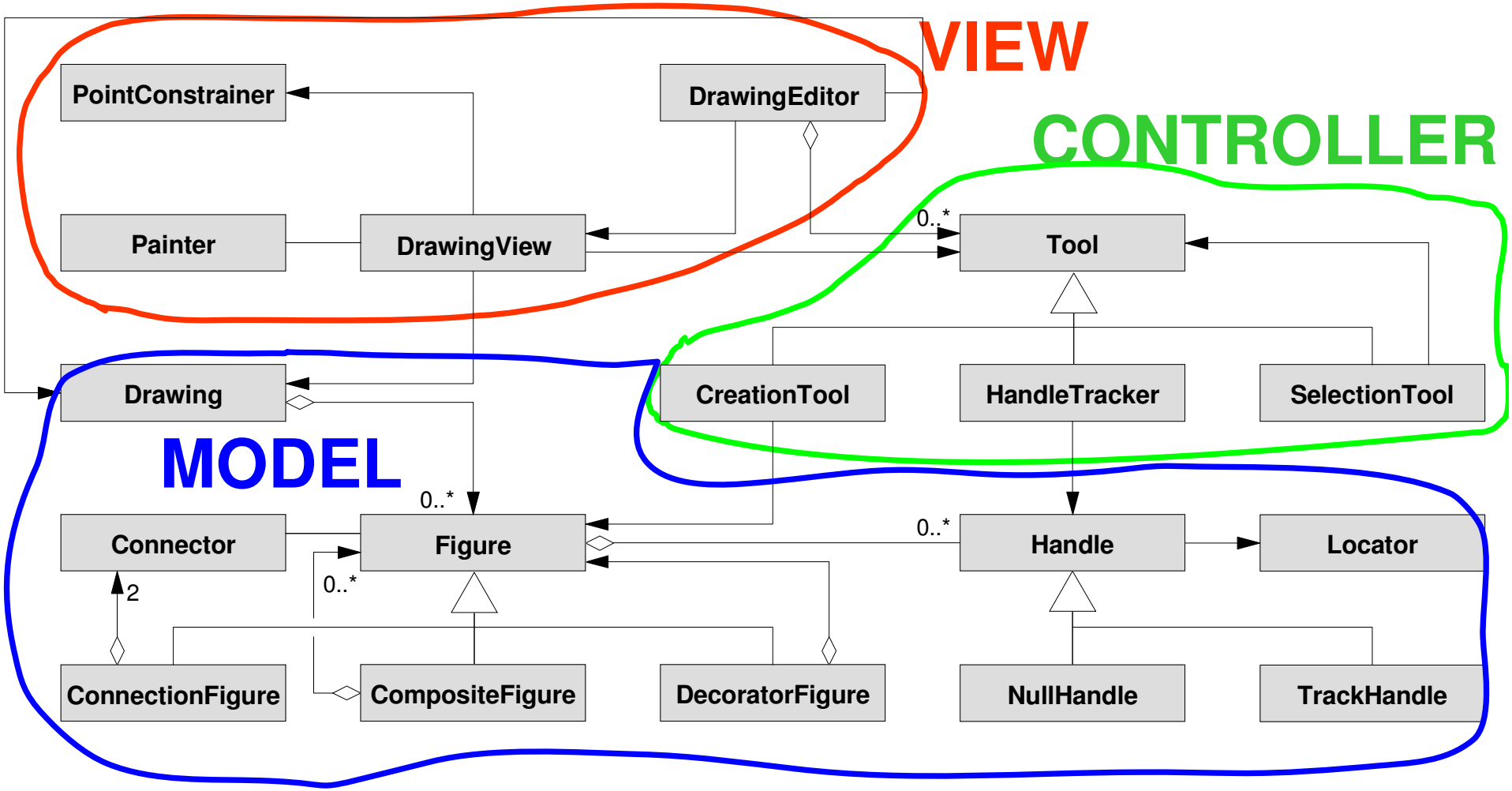
Challenge: maintain abstraction in Instantiation-Based View (IBV)

- Prevents excessive merging
  - Reduces abstraction
  - Leads to clutter
- Example: JHotDraw
  - 16,000 lines of Java
  - 200 classes
  - Rich inheritance hierarchy
  - Designed by object-oriented design experts

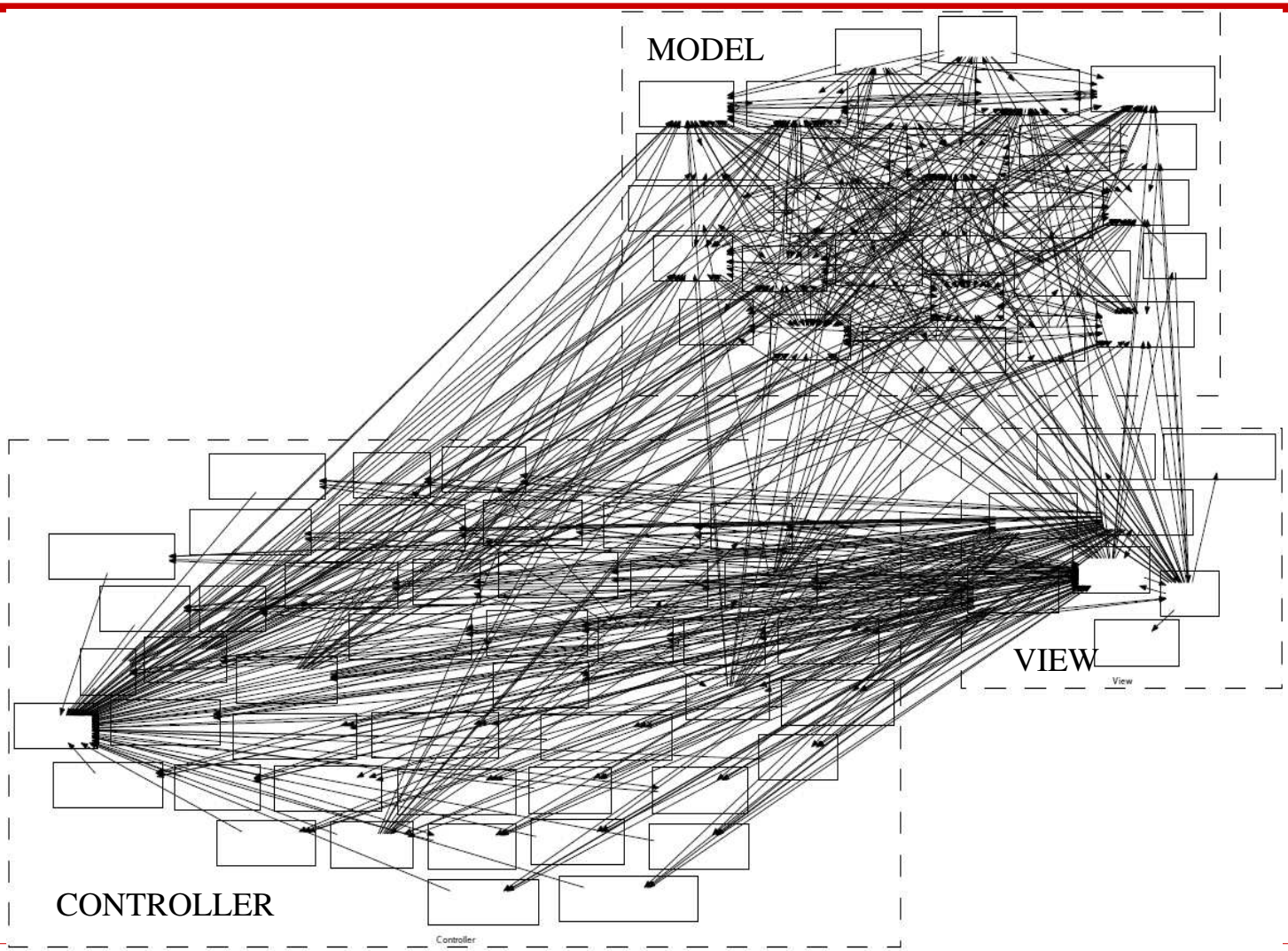
JHotDraw class diagram



Adding annotations to JHotDraw



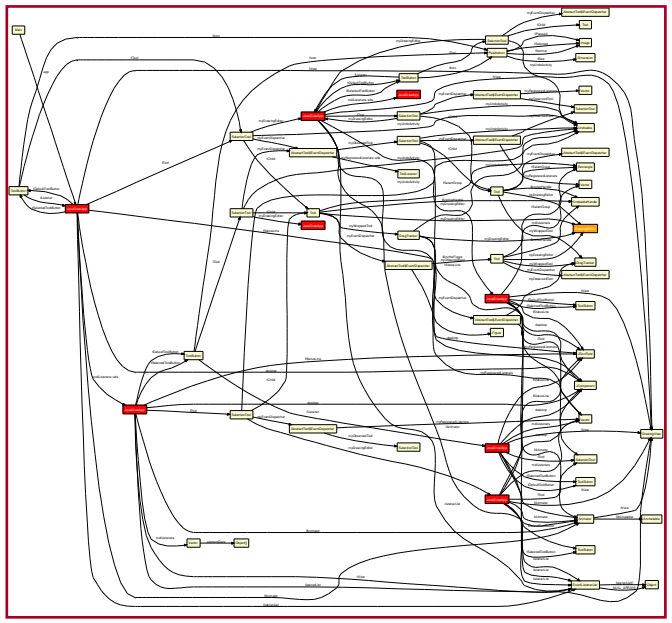
JHotDraw Object Graph (IBV)



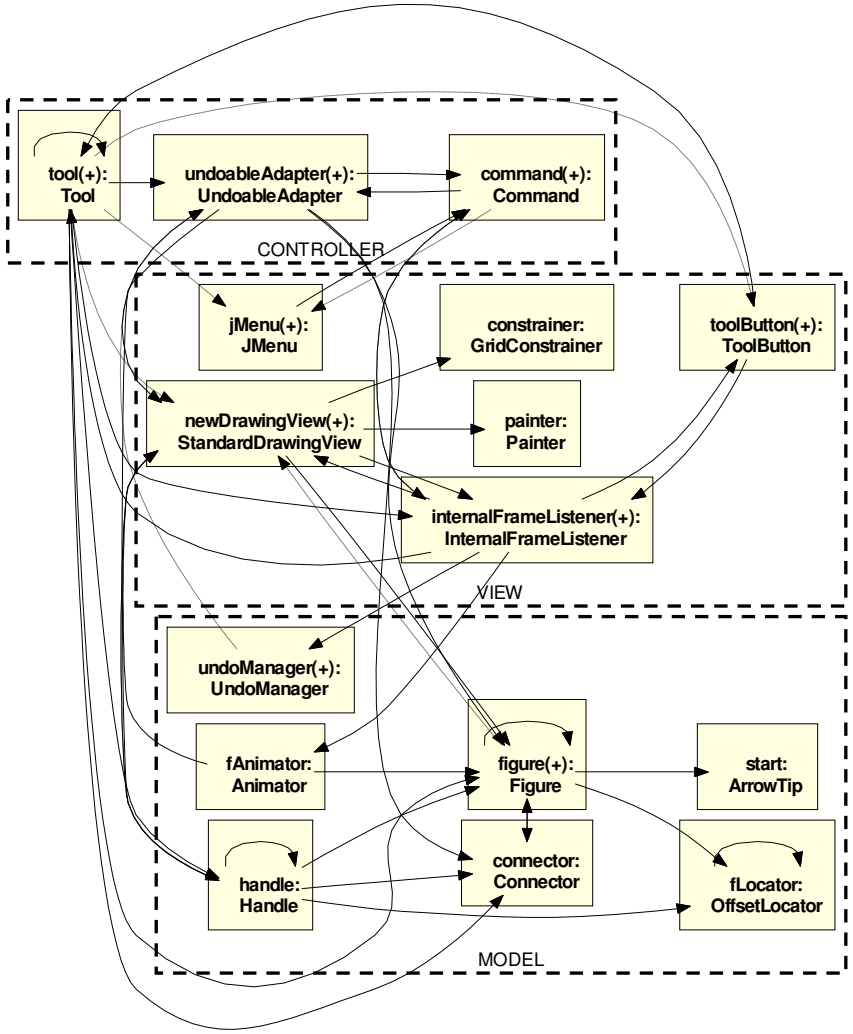
Abstraction by types

- (Optionally) Merge objects, **in a given domain**, based on their declared types
  - Heuristic (can be turned off)
  - Need not apply globally
- Abstract by **trivial types**
  - Merge objects when their types have **non-trivial least upper bound**
  - User configures list of trivial types. By default, includes `Object`, `Serializable`, etc.
- Or abstract by **design intent types**
  - Based on a list of architecturally relevant types, e.g., `Drawing, Figure, …`
  - Works well if type structure carefully designed
- Details in paper

JHotDraw with abstraction by types



Output of WOMBLE
[Jackson and Waingold, TSE'01]

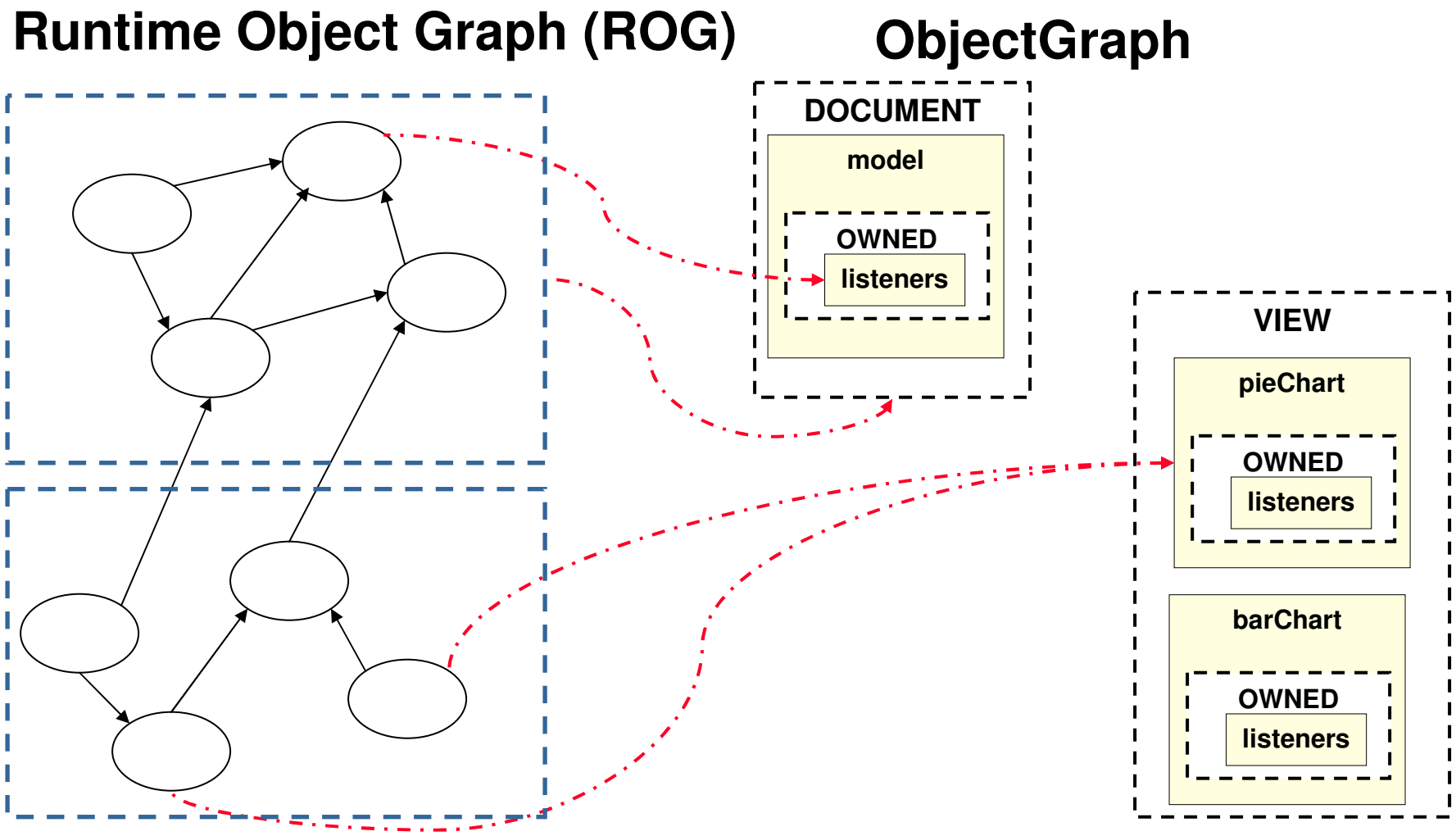


Soundness

Soundness

- Dynamic analysis previously used to infer runtime ownership structure for one or more program runs
- **Static analysis can** describe all possible program runs:
  - For **soundness, show** all objects and relations that may exist in any run
  - Maintain **aliasing soundness**: no one object should appear as two "boxes" in object graph

Intuition behind soundness



Theorem: Object and Domain Soundness

- Each object in a ROG has **exactly one representative** in an ObjectGraph.
- Similarly, each domain in the ROG has exactly one representative in the ObjectGraph.
- Furthermore, mapping is consistent with respect to the ownership relation.

Proving soundness

- Formalization using rewriting rules

Object Rules $\boxed{\Theta \implies \Theta'}$

$$\frac{\mathsf{AbstractObject}(o,\ d,\ t,\ \{b\ldots\})}{\Theta \vdash \mathsf{try}(\{o\},\ d)}\ [\text{R-CONVERT-OBJECT}]$$

$$\frac{\begin{array}{c}\mathsf{RuntimeObject}(\{o_{pull}\ldots\},\ d_{param}) \in \Theta \qquad \mathsf{RuntimeObject}(\{o_{parent}\ldots\},\ d_{parent}) \in \Theta \\ \mathsf{AbstractDomain}(d_{param}, \mathsf{typeof}(o_{parent})) \qquad \mathsf{aparam}(o_{parent}, d_{param}, d_{actual})\end{array}}{\Theta \vdash \mathsf{try}(\{o_{pull}\ldots\},\ d_{actual})}\ [\text{R-PULL-OBJECT}]$$

$$\frac{\Theta \vdash \mathsf{try}(\{o\ldots\},\ d) \qquad \nexists\, o, o_1.(\ \mathsf{RuntimeObject}(\{o_1\ldots\},\ d) \in \Theta\ \wedge\ \vdash \mathsf{compat}(\mathsf{typeof}(o),\ \mathsf{typeof}(o_1))\ )}{\Theta \implies \Theta, \mathsf{RuntimeObject}(\{o\ldots\},\ d)}\ [\text{R-NEW-OBJECT}]$$

$$\frac{\Theta \vdash \mathsf{try}(\{o\ldots\},\ d) \qquad \mathsf{compat}(\mathsf{typeof}(o),\ \mathsf{typeof}(o_1))}{\Theta, \mathsf{RuntimeObject}(\{o_1\ldots\},\ d) \implies \Theta, \mathsf{RuntimeObject}(\{o\ldots\} \cup \{o_1\ldots\},\ d)}\ [\text{R-MERGE-OBJECTS}]$$

$$\frac{\mathsf{compat}(\mathsf{typeof}(o_1),\ \mathsf{typeof}(o_2))}{\Theta, \mathsf{RuntimeObject}(\{o_1\ldots\},\ d), \mathsf{RuntimeObject}(\{o_2\ldots\},\ d) \implies \Theta, \mathsf{RuntimeObject}(\{o_1\ldots\} \cup \{o_2\ldots\}, d)}\ [\text{R-MERGE-EXISTING}]$$

Auxiliary Rules

$$\frac{\mathsf{AbstractObject}(o,\ d,\ t,\ \{d_{param} \mapsto d_{actual}, \ldots\})}{\mathsf{aparam}(o, d_{param}, d_{actual})} \qquad \frac{\mathsf{AbstractObject}(o,\ d,\ t,\ \{b\ldots\})}{\mathsf{typeof}(o) = t}$$

$$\mathsf{compat}(t_1,\ t_2) \text{ iff } t_1 <: t_2 \text{ or } t_2 <: t_1 \text{ or } \mathsf{existsNonTrivialLUB}(t_1,\ t_2) \text{ or } \mathsf{mapToSameDIT}(t_1,\ t_2) \quad [\text{R-AUX-COMPAT}]$$

Proof of Object and Domain Soundness

- Ownership domains formalization
  - Featherweight Domain Java
  - Store typing characterizes any execution of a well-typed program
- Soundness proof relates store typing to extracted ObjectGraph

Future work: Edge Soundness

- Edges in an ObjectGraph soundly abstract all field points-to relations between objects in an ROG.
- If object ℓ_1 has a field reference to object ℓ_2 in a ROG, then there is an edge between the representatives of ℓ_1 and ℓ_2 in the ObjectGraph.

Evaluation

Evaluated on several case studies and a field study (totaling 68 KLOC)

- Research Question: *does an extracted ObjectGraph suffer from too much or too little abstraction on real code?*

System	Size	Comments
JHotDraw	15 KLOC	Designed by object-oriented design experts
HillClimber	15 KLOC	Designed by undergraduates
Aphyds	8 KLOC	Original developer drew architecture
LbGrid	30 KLOC	Part of 250-KLOC commercial system

Related work

- Object graph extraction
  - Without using annotations
    [Jackson and Waingold, ICSE'99,TSE'01]
    [O'Callahan, Ph.D. thesis'01]
  - Using non-ownership annotations
    [Lam and Rinard, ECOOP'03]
  - Some unsound w.r.t. aliasing or inheritance
  - Non-hierarchical object models
- Related static analyses
  - Points-to analysis [e.g., Milanova et al., TOSEM'05]
  - Shape analysis [e.g., Sagiv et al., POPL'99]
  - Precise, but non-hierarchical abstractions

Related work (continued)

- Radical language extensions
  - Specify architectural hierarchy and instances
    ArchJava [Aldrich et al., ECOOP'02]
    JCoBox [Schäfer et al., CoCoME'07]
  - Restrict passing object references
  - Require re-engineering existing systems
- Dynamic ownership analyses
  - Organize objects based on ownership
    [Hill, Noble and Potter, J. Vis. Lang.& Comp.,'02]
  - Infer strict-as-dominator hierarchy
  - Describe structure for few program runs

Summary

- **Static analysis** to extract **hierarchical object graph**
  - Relies on **ownership type annotations**
  - Proved **object/soundness** theorem
- **Evaluated analysis** on real object-oriented code totaling 68 KLOC
  - Achieves adequate precision
  - Sufficient to relate abstracted object graph to a posited target architecture [Tech. report]