A Field Study in Static Extraction of Runtime Architectures

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

Acknowledgements

SIGSOFT CAPS *for travel funding*

LogicBlox Inc.

Brad A. Myers William Scherlis

Thomas LaToza Christopher Scaffidi

Object-Oriented Code vs. Runtime Structure

*"An object-oriented program's **runtime structure** often bears little resemblance to its **code structure**.*

*The **code structure** [...] consists of **classes in fixed inheritance relationships**.*

*A program's **runtime structure** consists of [...] **networks of communicating objects** [...]*

Trying to understand one from the other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals, and vice versa." (Gamma et al., 1994)

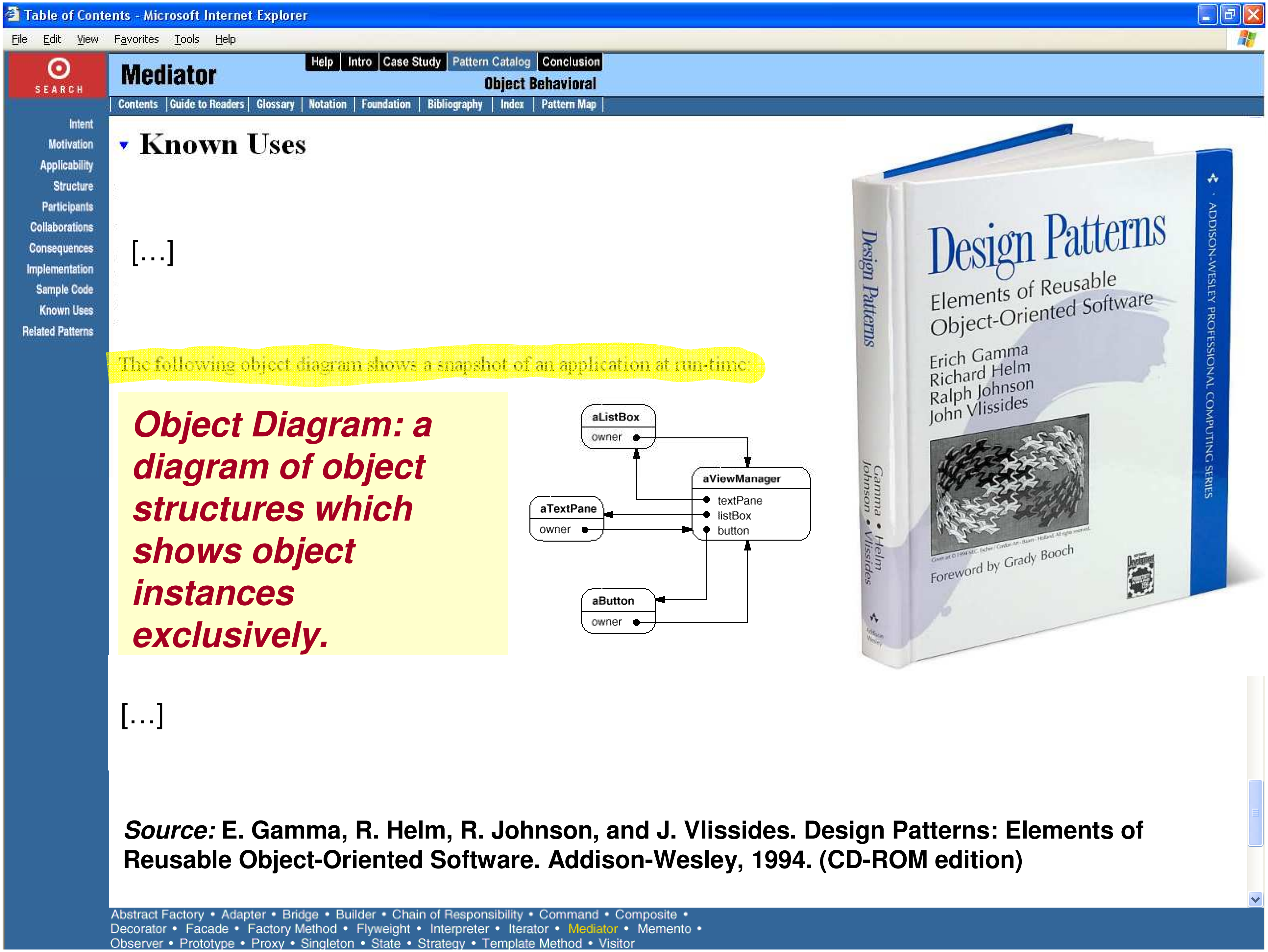


Table of Contents - Microsoft Internet Explorer
File
Edit
View
Favorites
Tools
Help
SEARCH
Help
Intro
Case Study
Pattern Catalog
Conclusion
Contents
Guide to Readers
Glossary
Notation
Foundation
Bibliography
Index
Pattern Map
Intent
Motivation
Applicability
Structure
Participants
Collaborations
Consequences
Implementation
Sample Code
Known Uses
Related Patterns
aListBox
owner
aViewManager
textPane
listBox
button
aTextPane
owner
aButton
owner
Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
Gamma • Helm • Johnson • Vlissides
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES
Abstract Factory • Adapter • Bridge • Builder • Chain of Responsibility • Command • Composite •
Decorator • Facade • Factory Method • Flyweight • Interpreter • Iterator • Mediator • Memento •
Observer • Prototype • Proxy • Singleton • State • Strategy • Template Method • Visitor

Mediator

Object Behavioral

▾ Known Uses

[…]

The following object diagram shows a snapshot of an application at run-time:

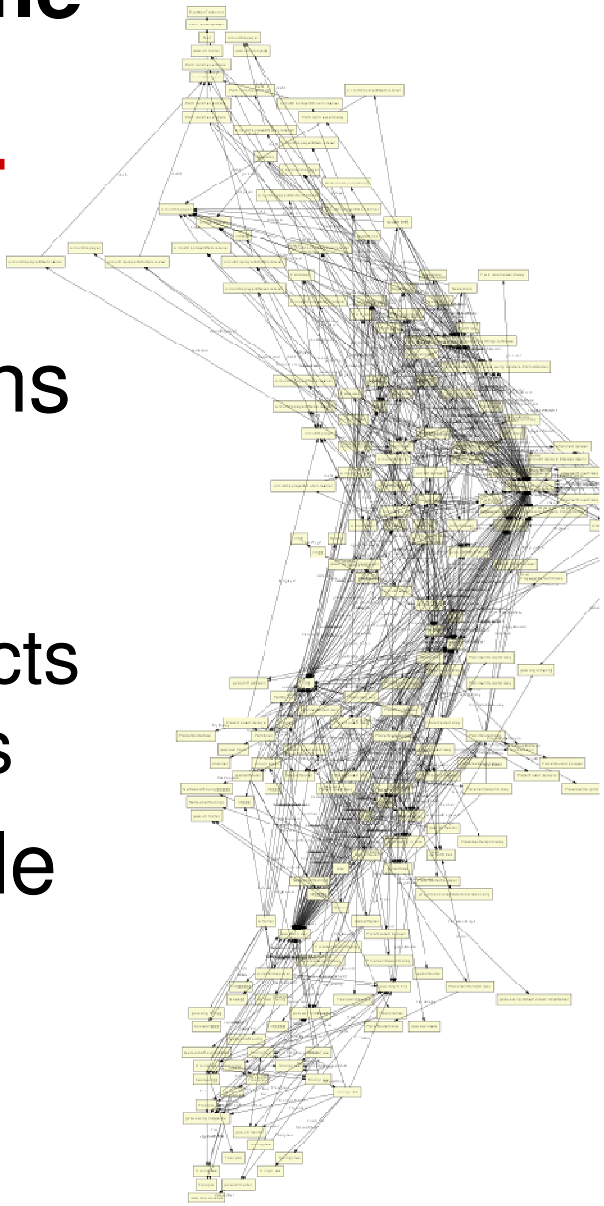
Object Diagram: a diagram of object structures which shows object instances exclusively.

[…]

Source: **E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994. (CD-ROM edition)**

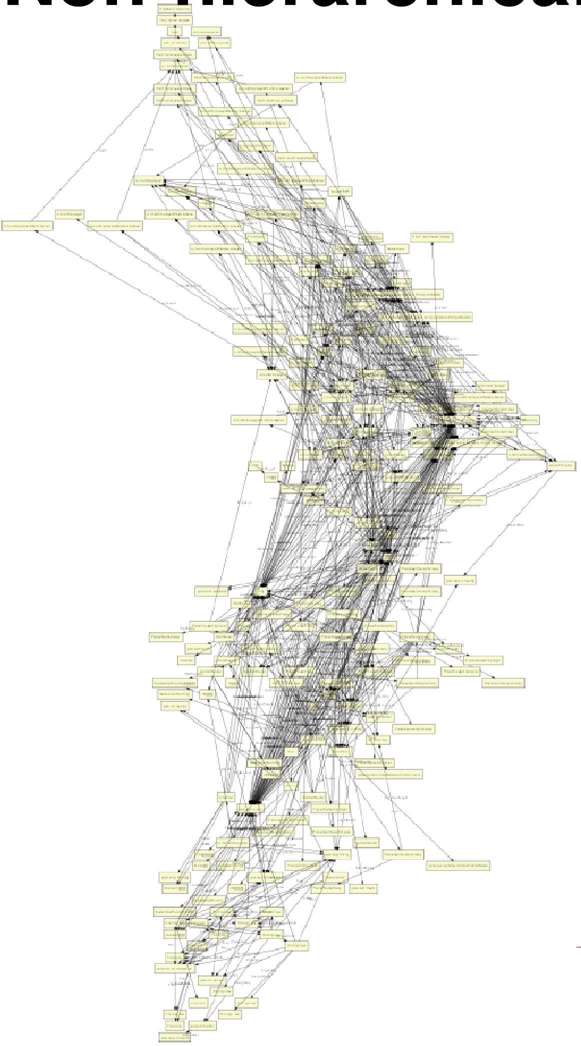
Tool support to extract runtime architecture still immature

- Show networks of objects
- Non-hierarchical object graphs
  - **No architectural abstraction**
  - Low-level objects mixed with architecturally significant objects
  - No scale-up to large programs
- Sometimes, incorrectly handle **aliasing, inheritance**

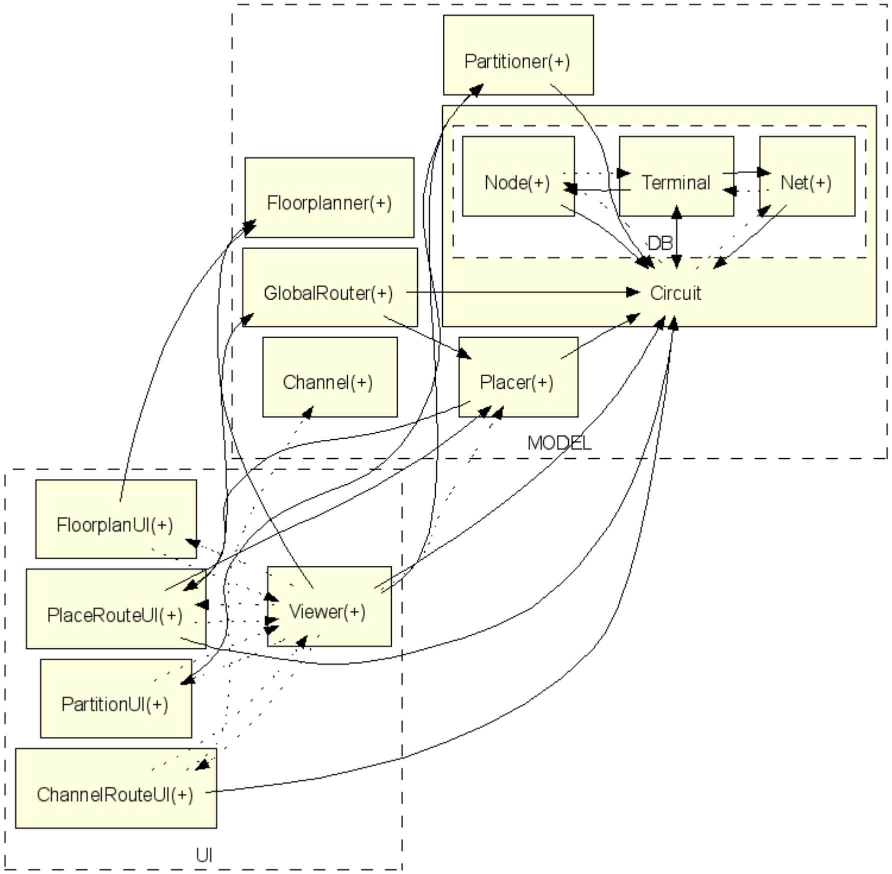


Abstraction through object hierarchy

Non-hierarchical graph



Hierarchical graph



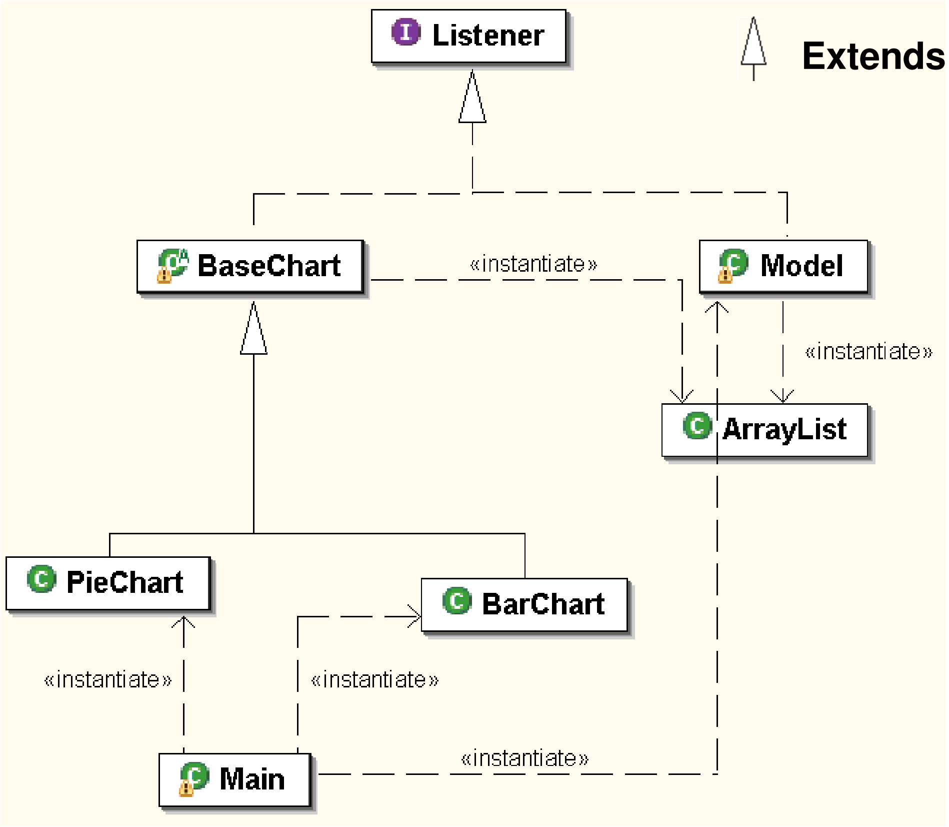
Key Insight

Ownership domain annotations enable the extraction of **sound hierarchical** object graphs using **static analysis**

Extracting sound hierarchical object graphs using static analysis

- **Static analysis**
  - Dynamic analysis can show object graphs for few program runs, not all possible ones
- **Sound**
  - Account for all objects and relations
  - That could exist in any program run
- **Hierarchical object graphs**
  - Provide **architectural abstraction**
  - Push low-level objects under more architecturally relevant objects

Many tools extract a code architecture

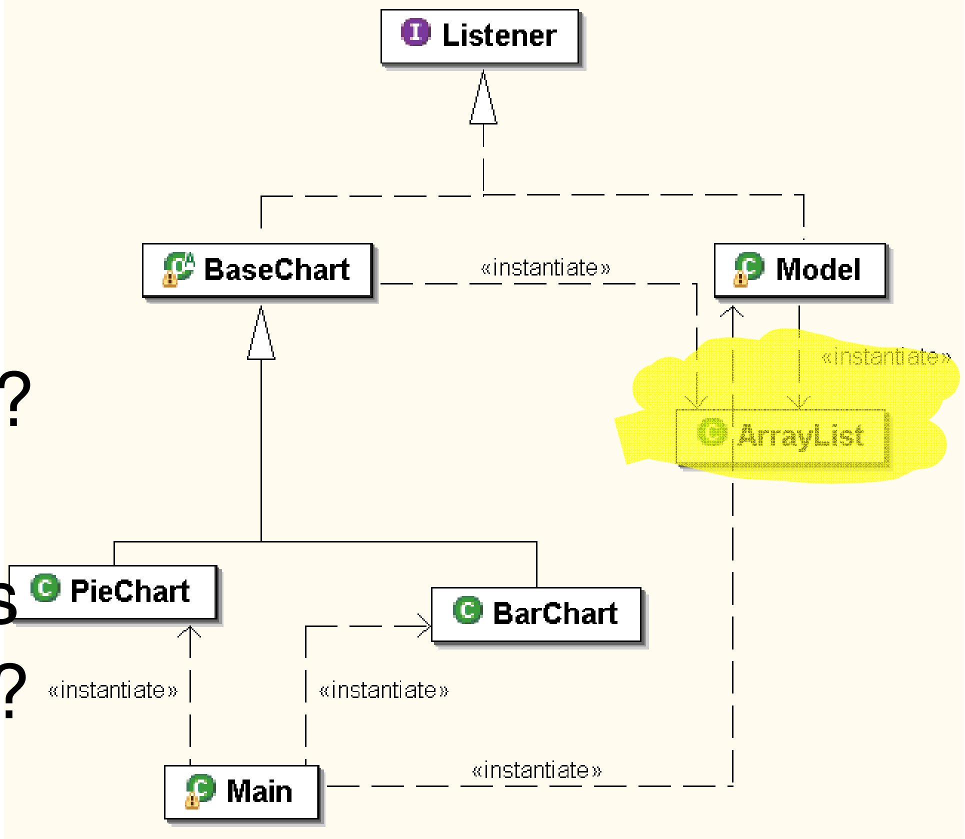


Class diagram extracted by Eclipse UML.

```
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;
}
```

Code architecture does not help explain several facts

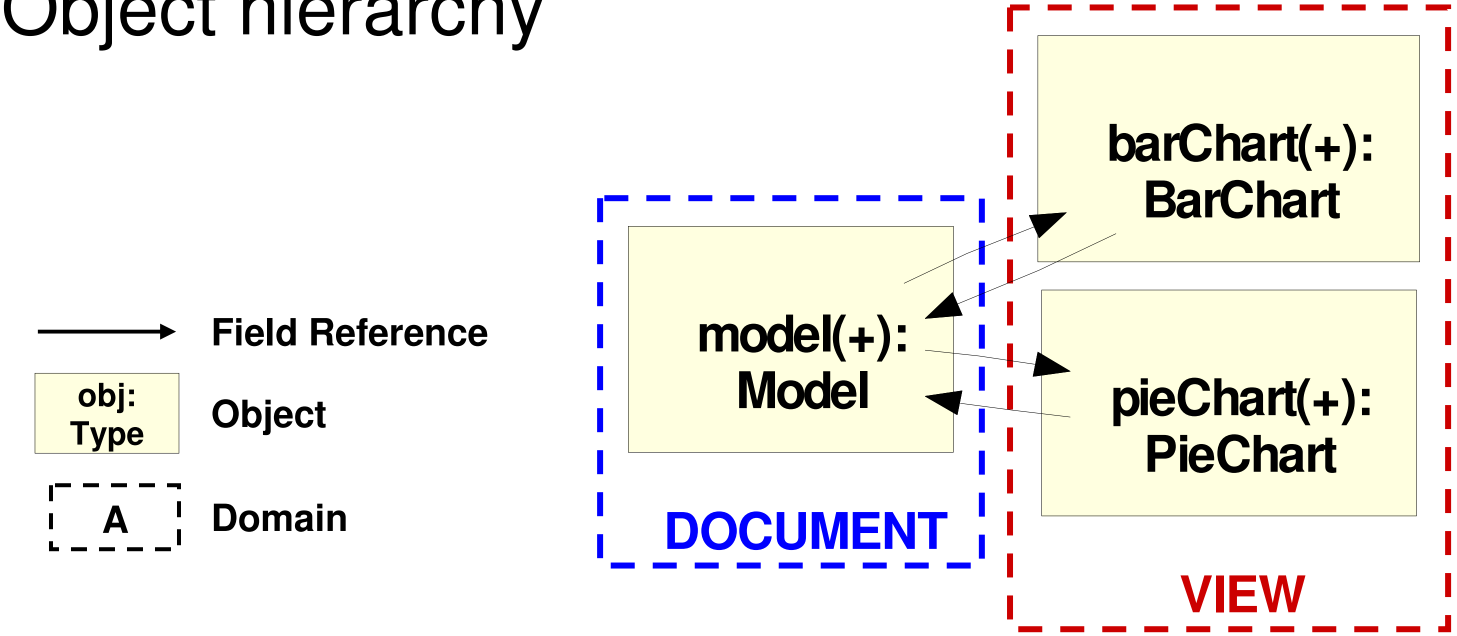
- Is this a Document-View architecture?
- Do `PieChart`, `BarChart`, `Model` share one `Listener`?
- Are different `ArrayList` instances conceptually different?



Class diagram extracted by Eclipse UML.

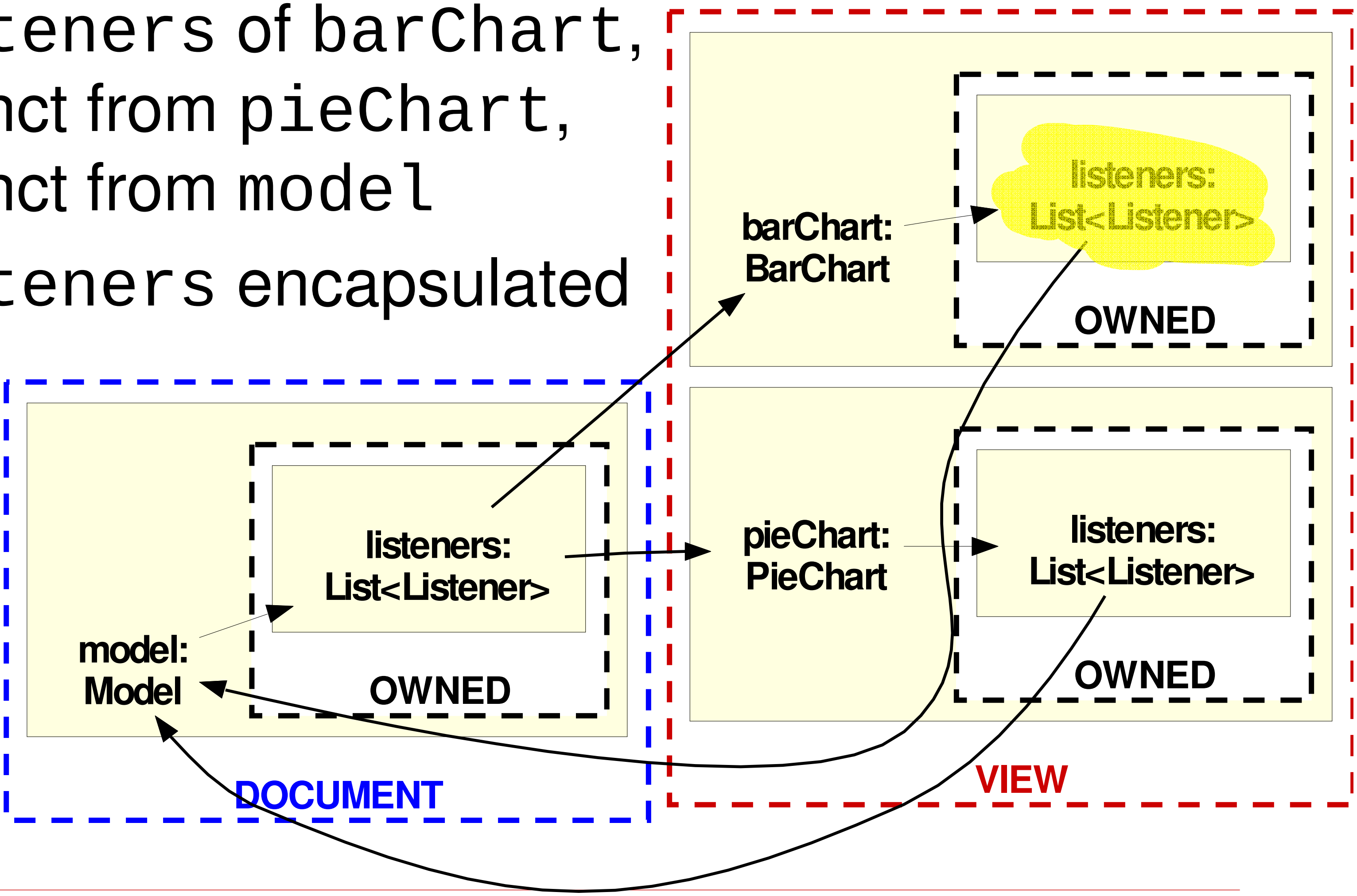
Hierarchical object graphs

- Show **objects**, instead of types
- Convey **architectural intent**
    - Domains = conceptual groups (or tiers)
    - Object hierarchy

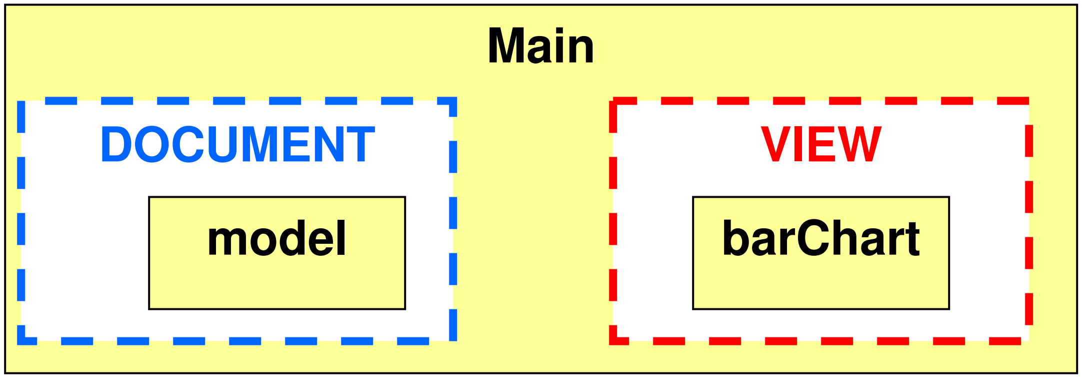


Hierarchy enables varying abstraction level

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



Ownership domain annotations
[Aldrich and Chambers, ECOOP'04]



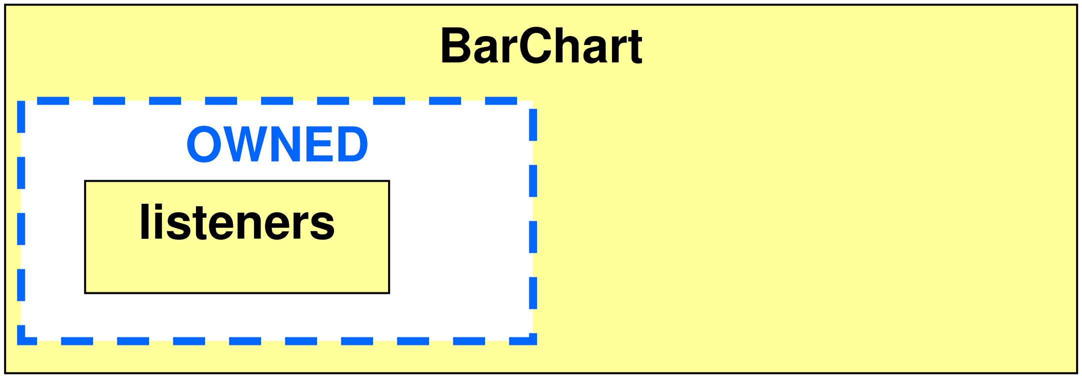
```
class Main {
  domain DOCUMENT, VIEW;

  DOCUMENT Model model;
  VIEW BarChart barChart;

  ...
}
```

Declarations are simplified

Domains can be defined at the top-level

Ownership domain annotations

[Aldrich and Chambers, ECOOP'04]



BarChart

OWNED

listeners

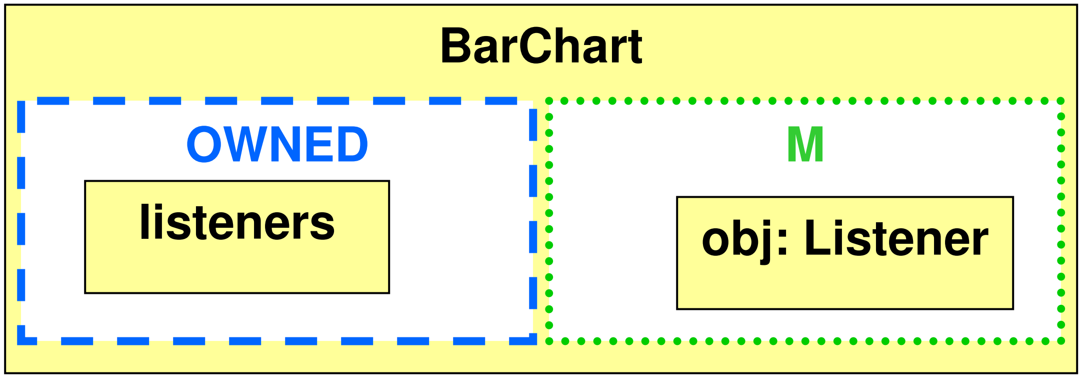
```
class BarChart {
  domain OWNED;
  OWNED List listeners;
  ...
}
```

Declarations are simplified

Domains can be declared inside each object

Ownership domain annotations

[Aldrich and Chambers, ECOOP'04]

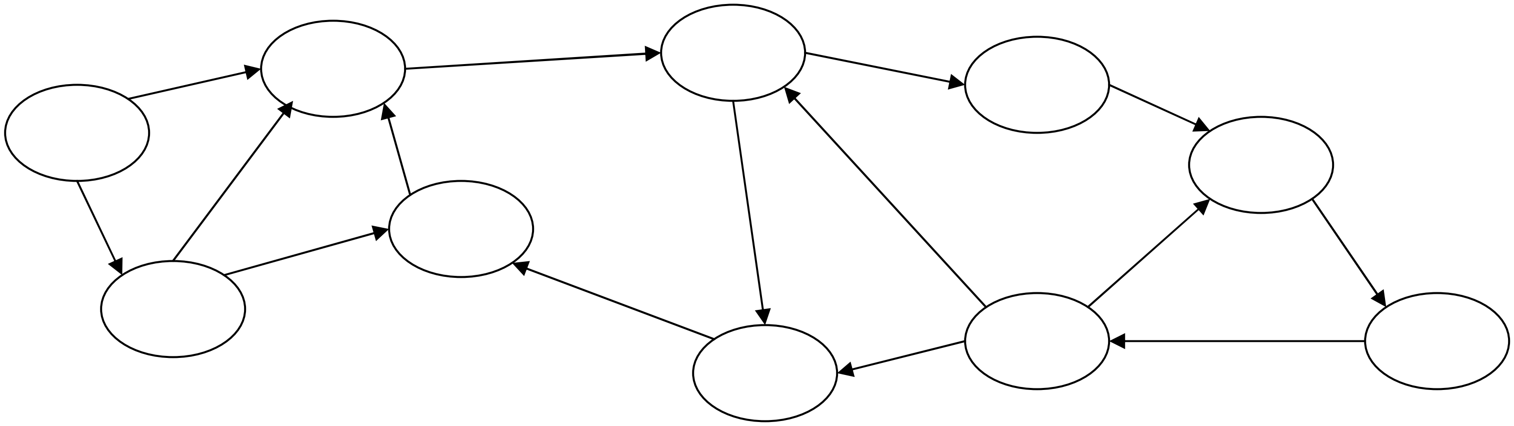


```
class BarChart < M > {
  domain OWNED;
  OWNED List< M Listener> listeners;
}
 ...
 VIEW BarChart<DOCUMENT> barChart;
```

Domain parameters allow sharing of state

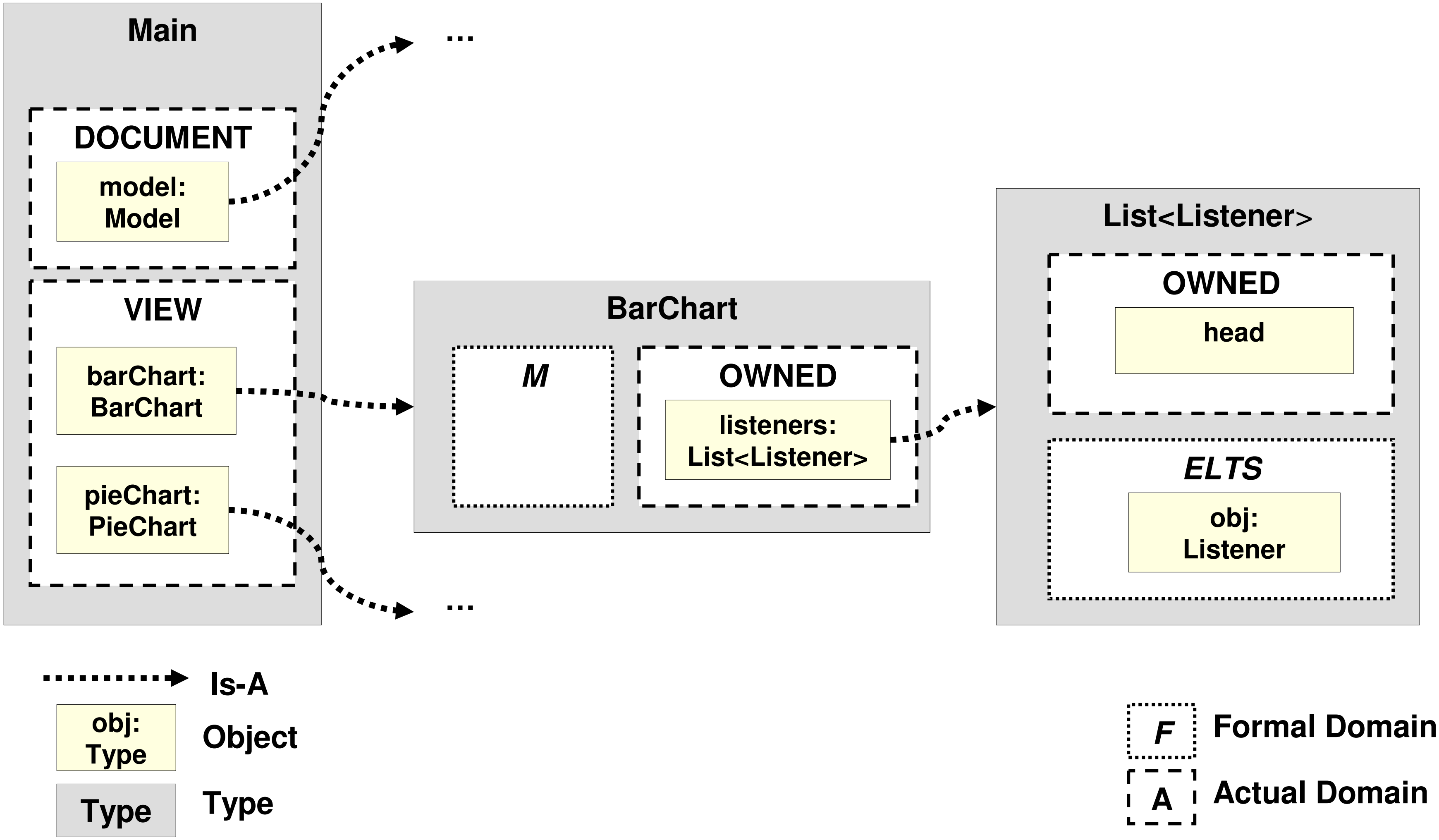
Static analysis: TypeGraph ObjectGraph

- Build **TypeGraph** from program’s AST
- Convert to **ObjectGraph** that soundly approximates all **runtime object graphs** (ROG)

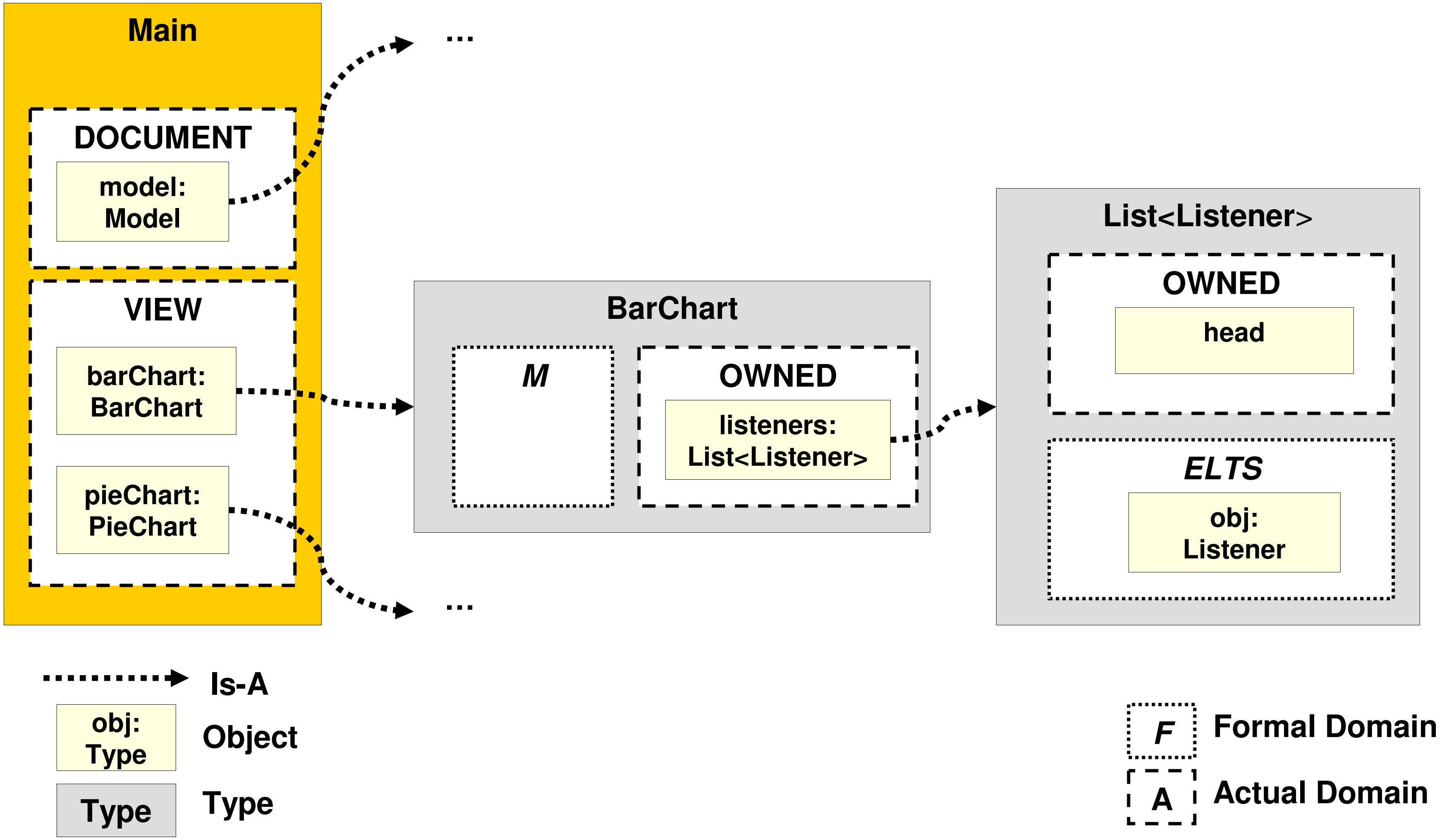


ROG: graph where nodes represent runtime objects, edges represent point-to relations

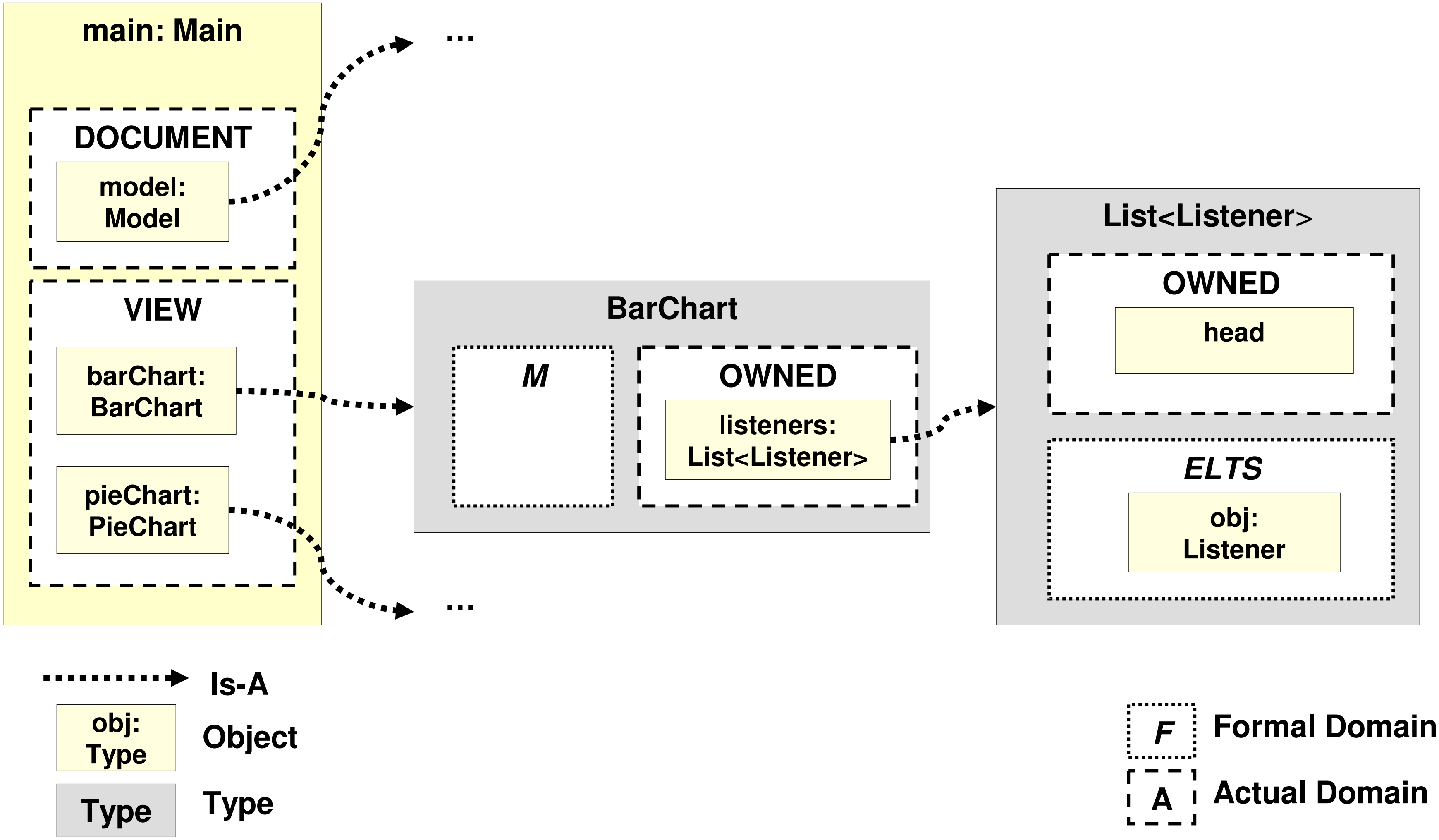
TypeGraph: show types, domains inside types, and objects in domains



ObjectGraph: instantiate types, starting with root

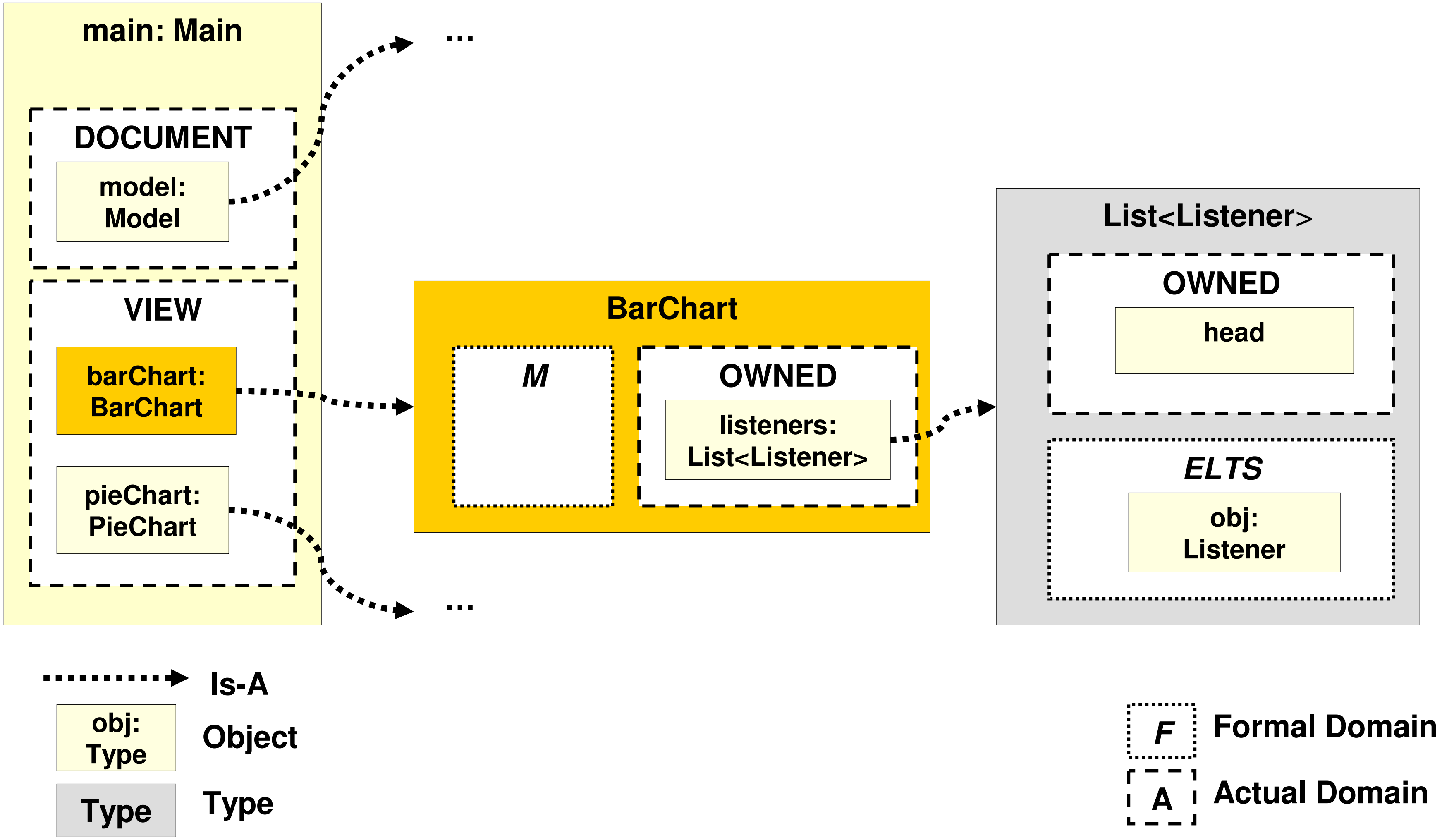


ObjectGraph: instantiate types, starting with root

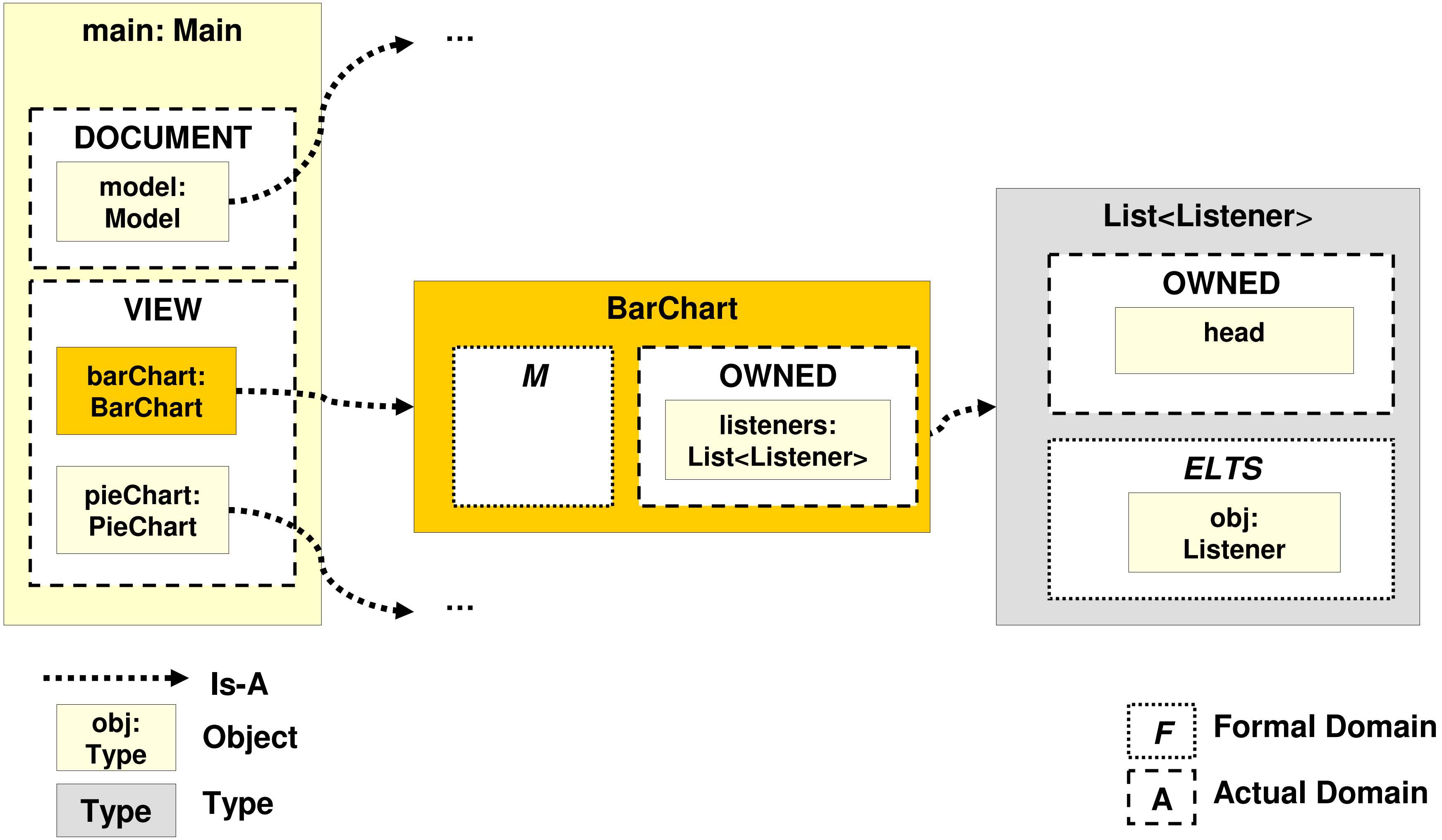


● Problem ● Approach ● **Analysis** ● Evaluation ● Conclusion

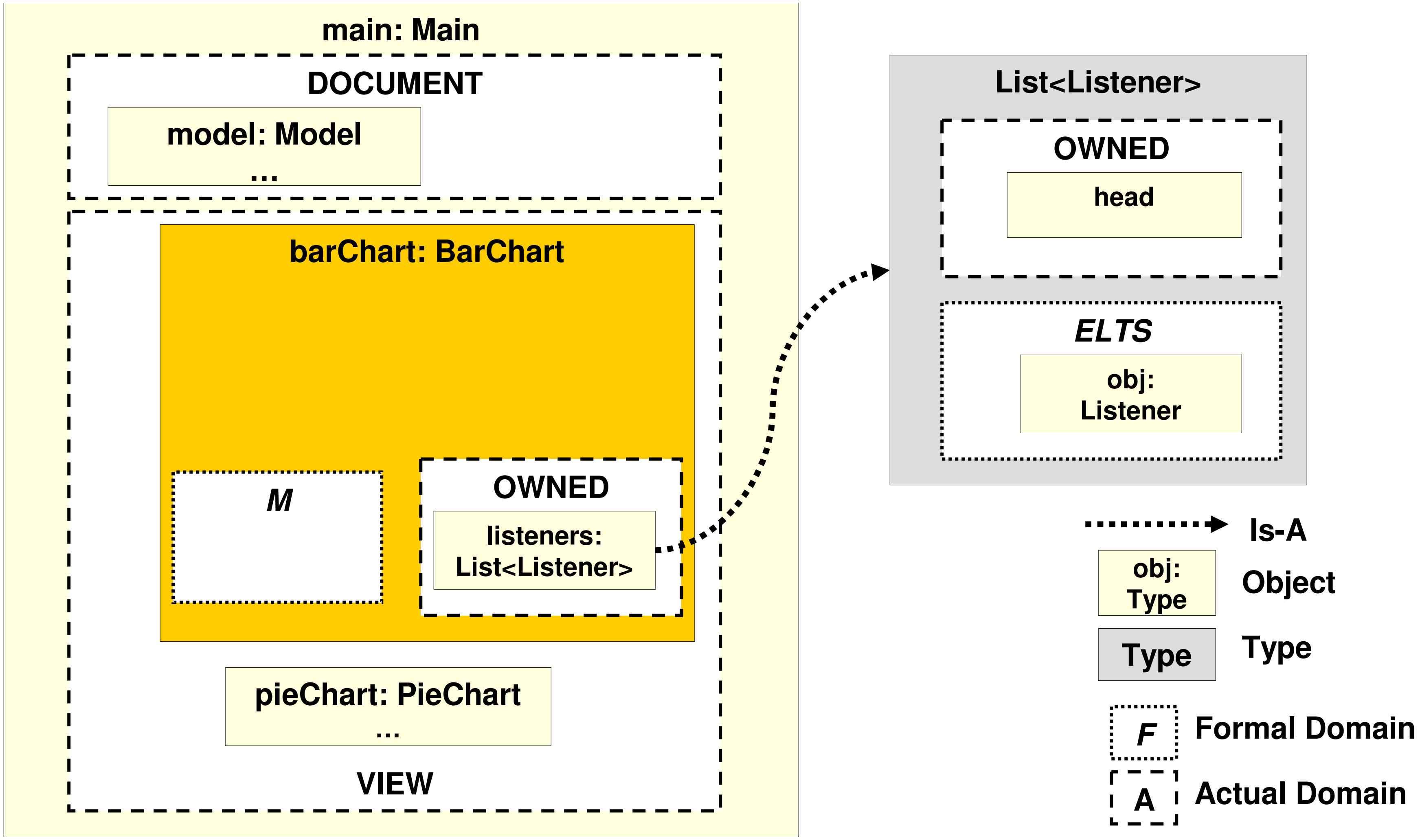
ObjectGraph: instantiate types, show domains and objects inside domains



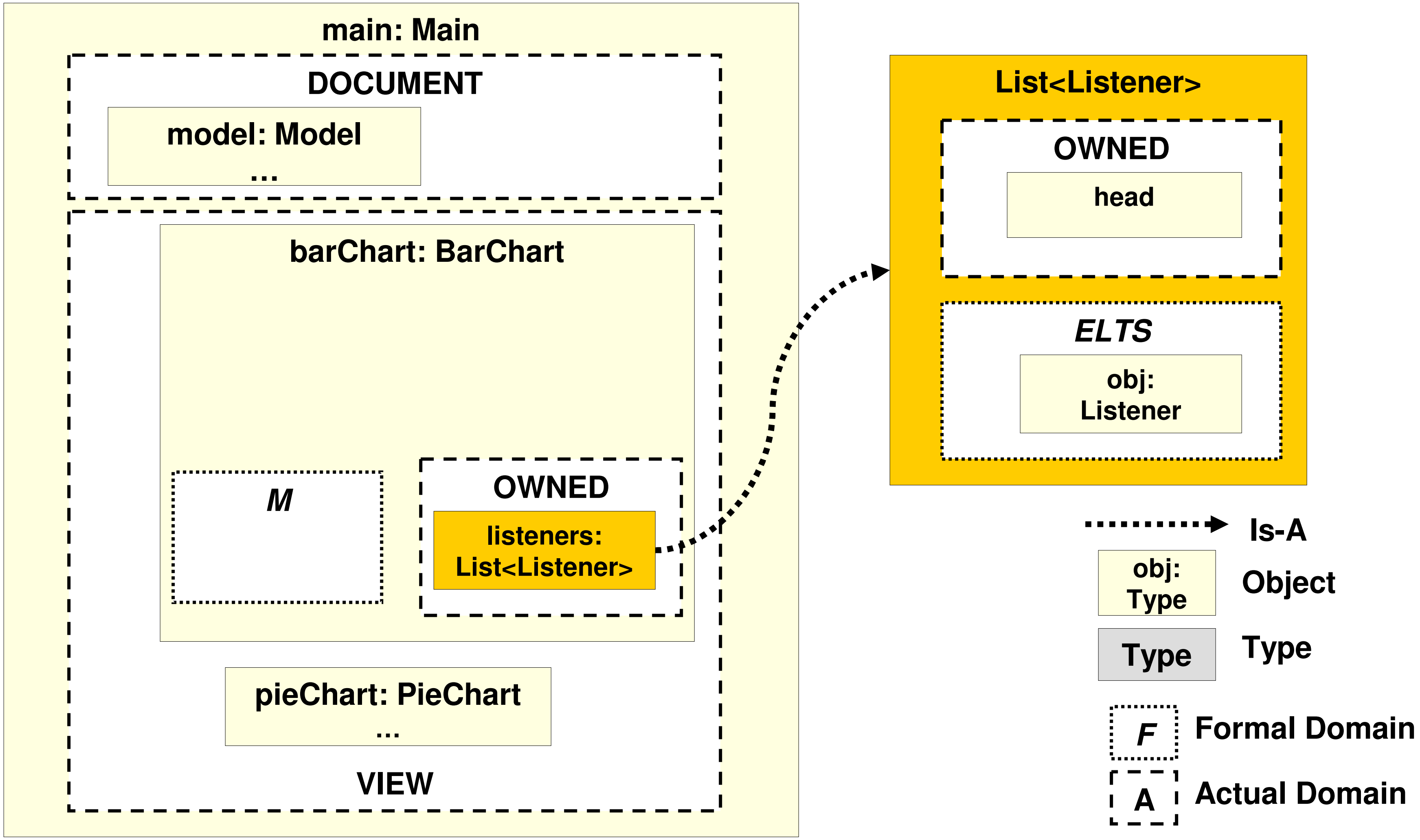
ObjectGraph: instantiate types, show domains and objects inside domains



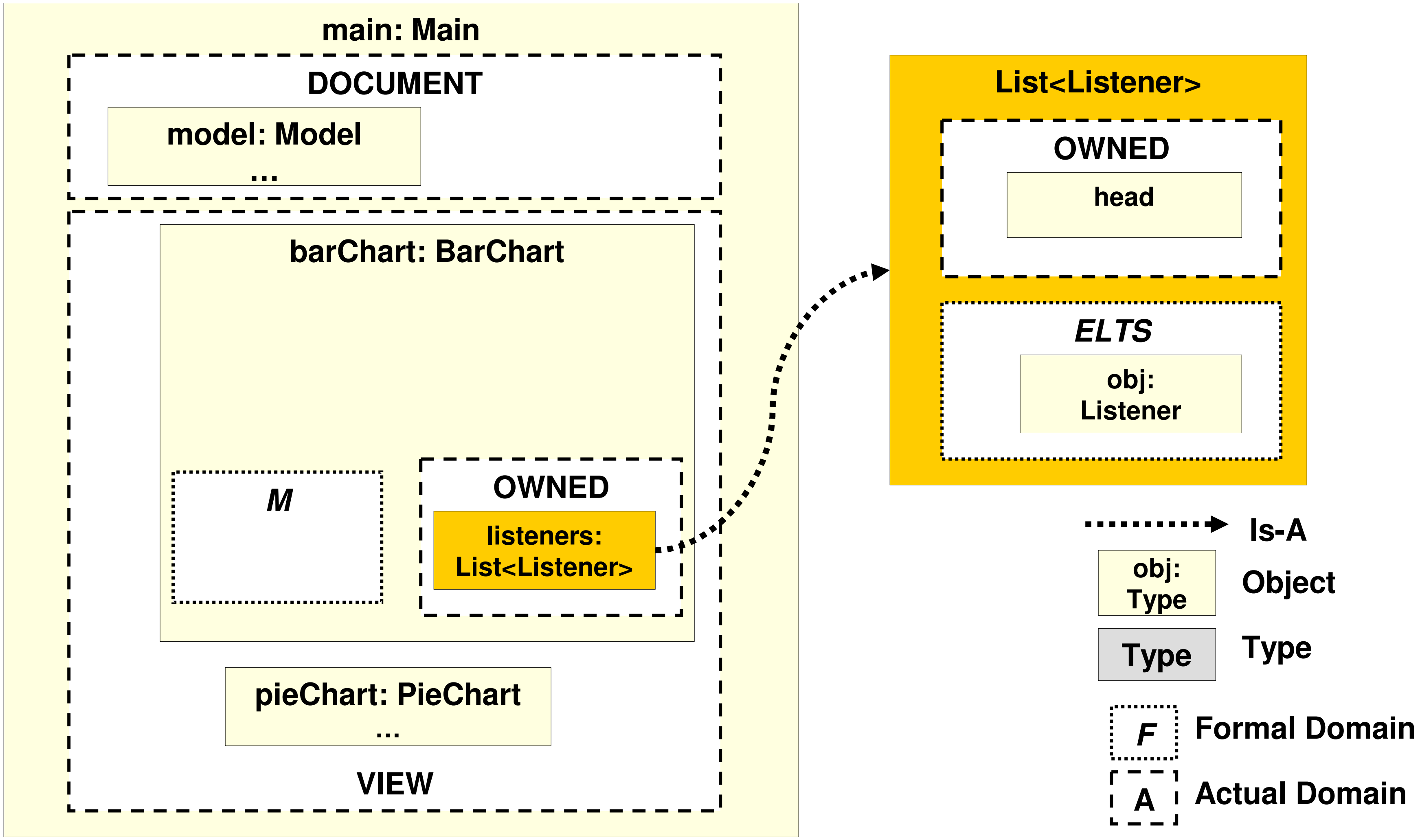
ObjectGraph: instantiate types, show domains and objects inside domains



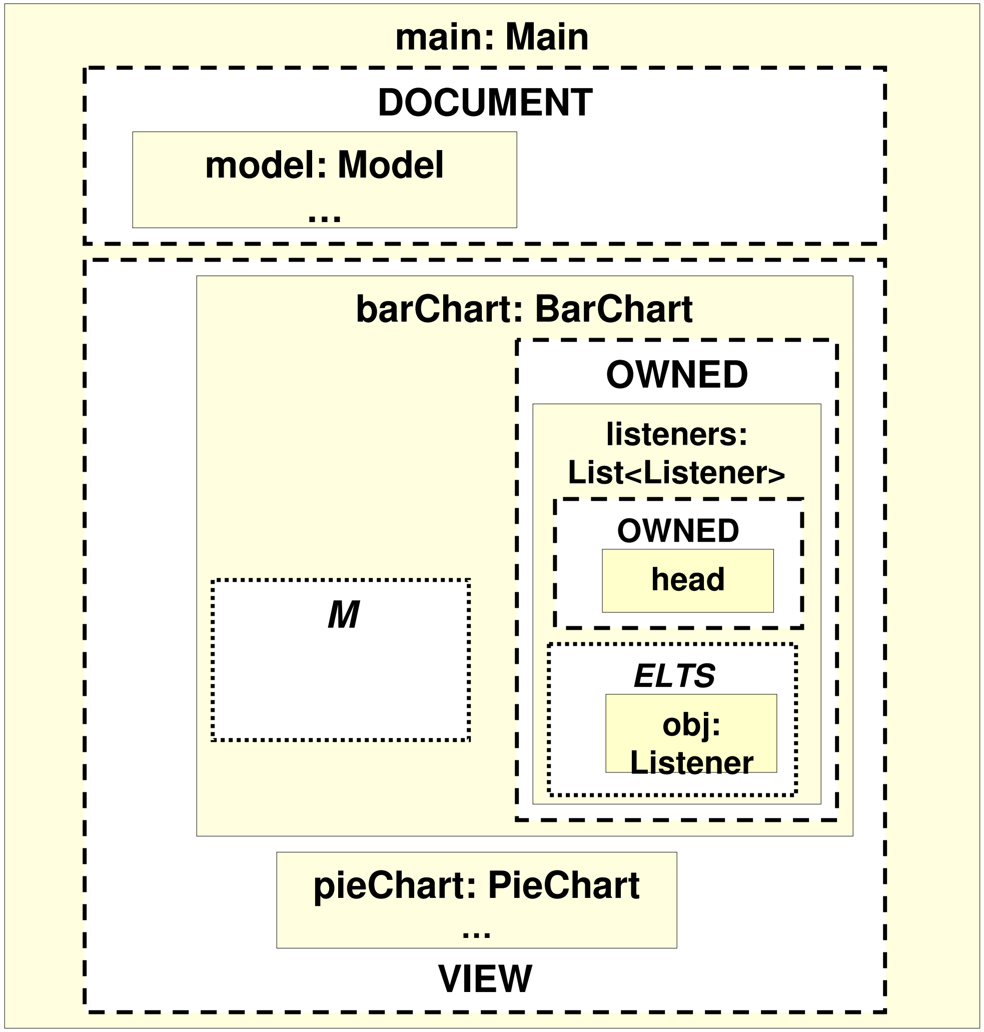
ObjectGraph: instantiate types, show domains and objects inside domains



ObjectGraph: instantiate types, show domains and objects inside domains



ObjectGraph: instantiate types, show domains and objects inside domains



main: Main

DOCUMENT

model: Model
...

barChart: BarChart

OWNED

listeners:
List<Listener>

OWNED

head

ELTS

obj:
Listener

M

pieChart: PieChart
...

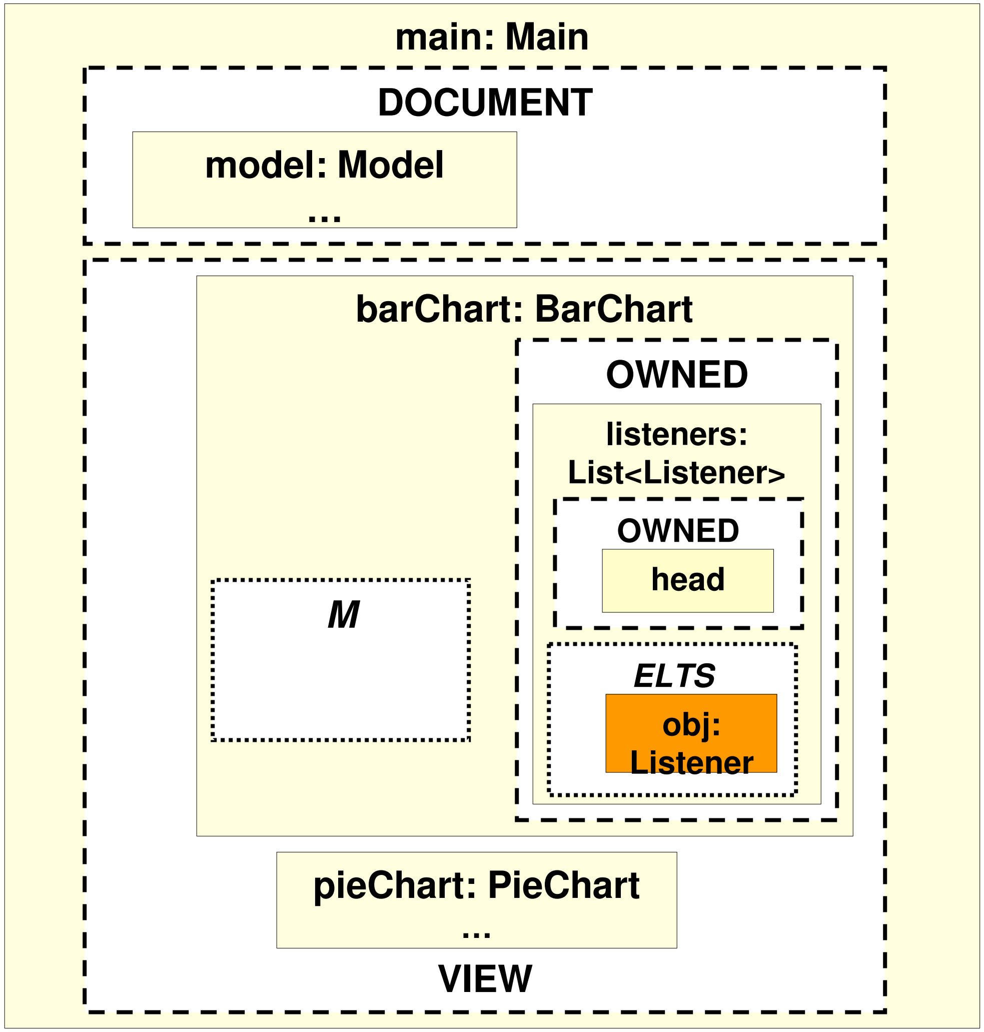
VIEW

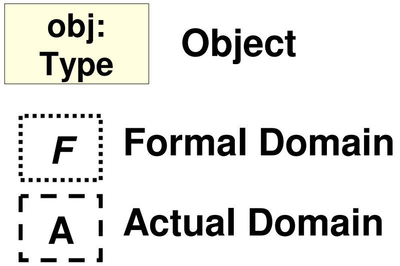
obj:
Type
Object

F Formal Domain

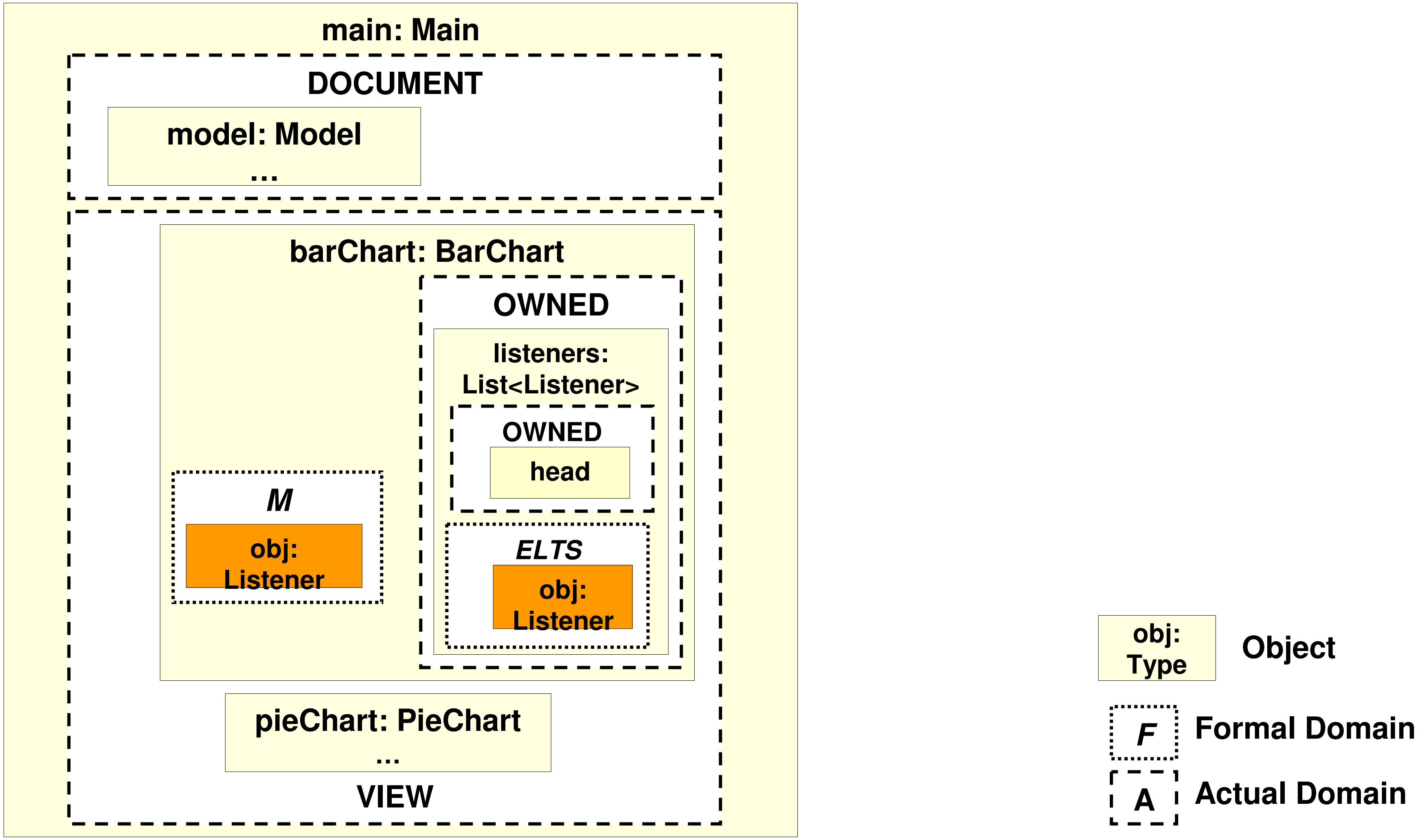
A Actual Domain

ObjectGraph: pull objects from formal domains to actual domains

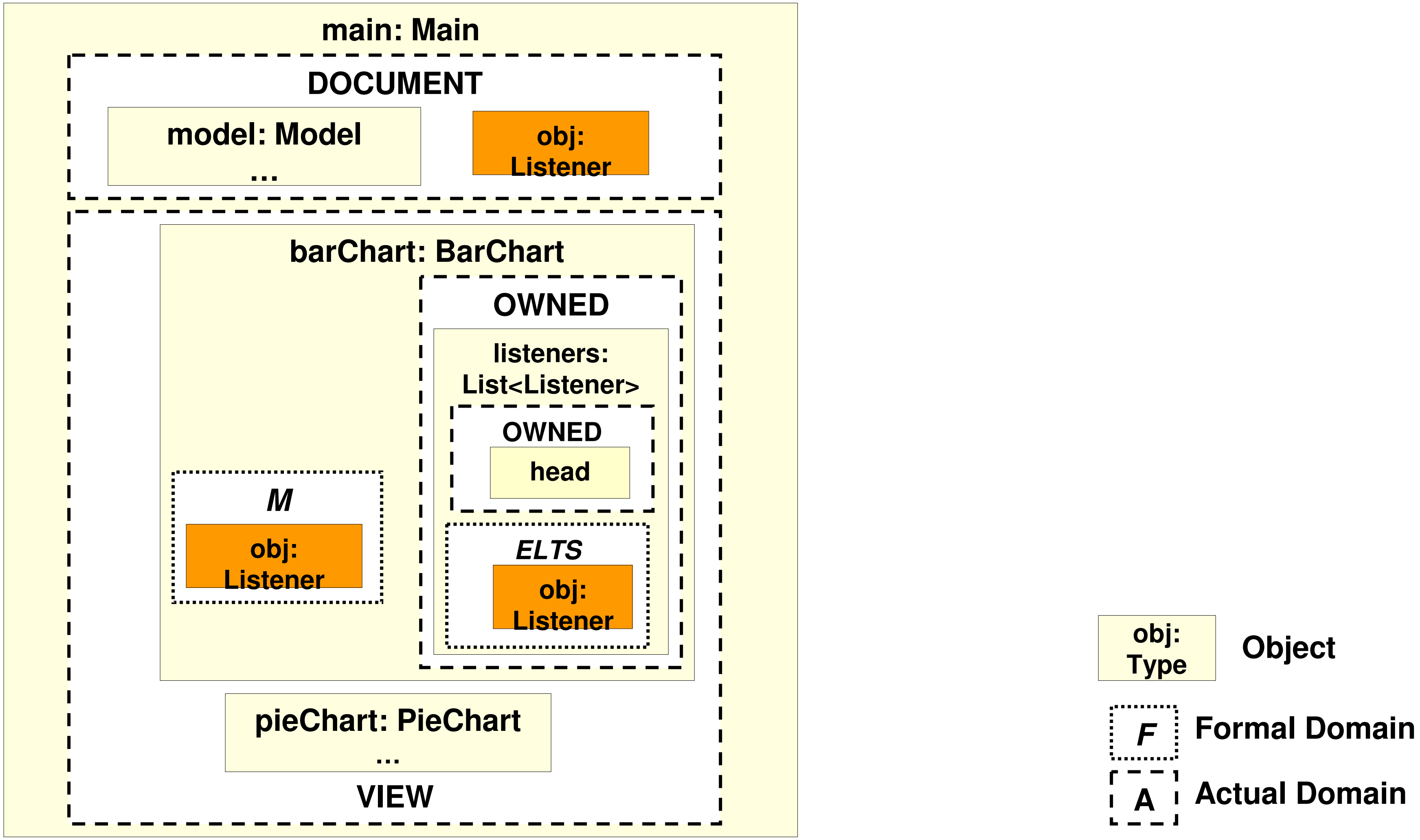




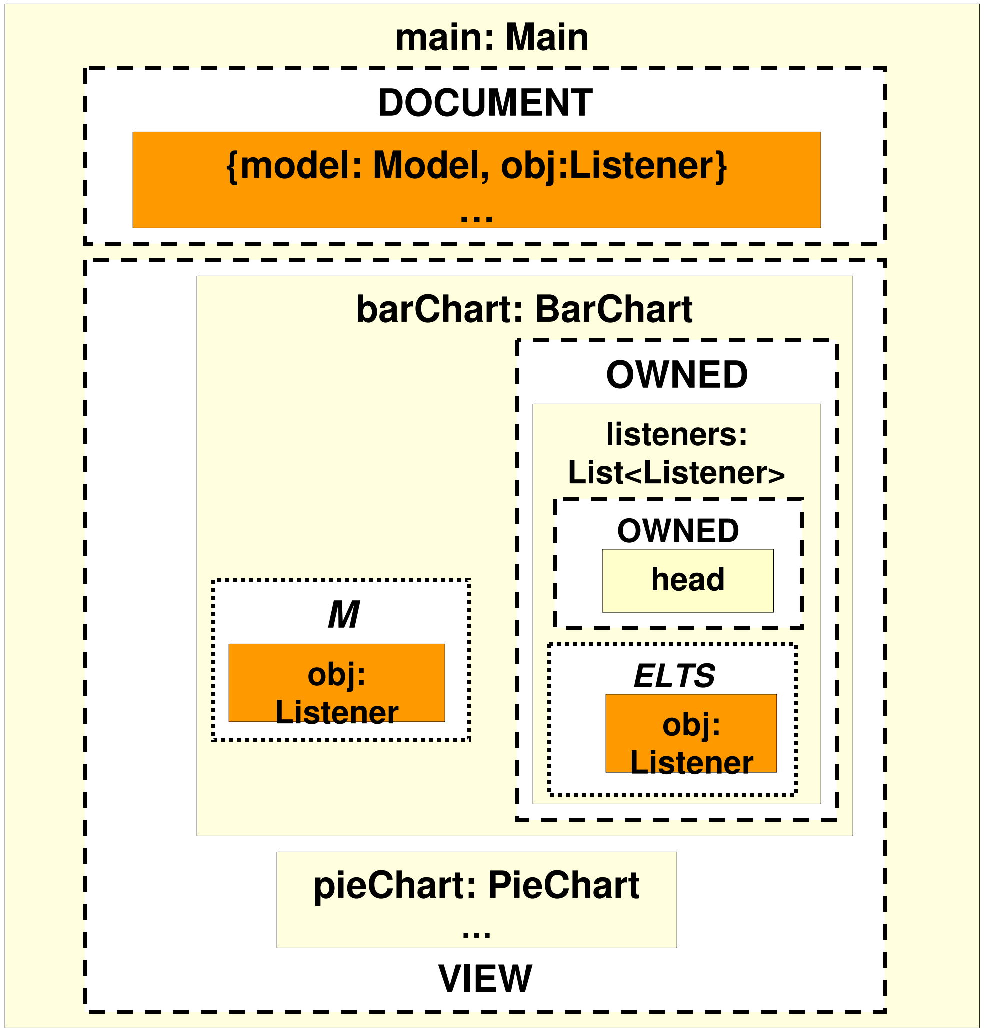
ObjectGraph: pull objects from formal domains to actual domains



ObjectGraph: pull objects from formal domains to actual domains



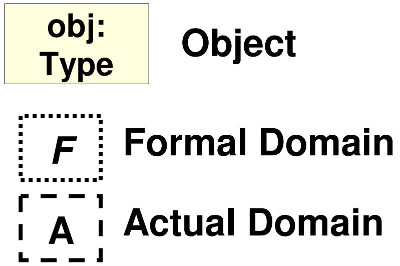
ObjectGraph: merge objects, in one domain, that *may* alias, based on types



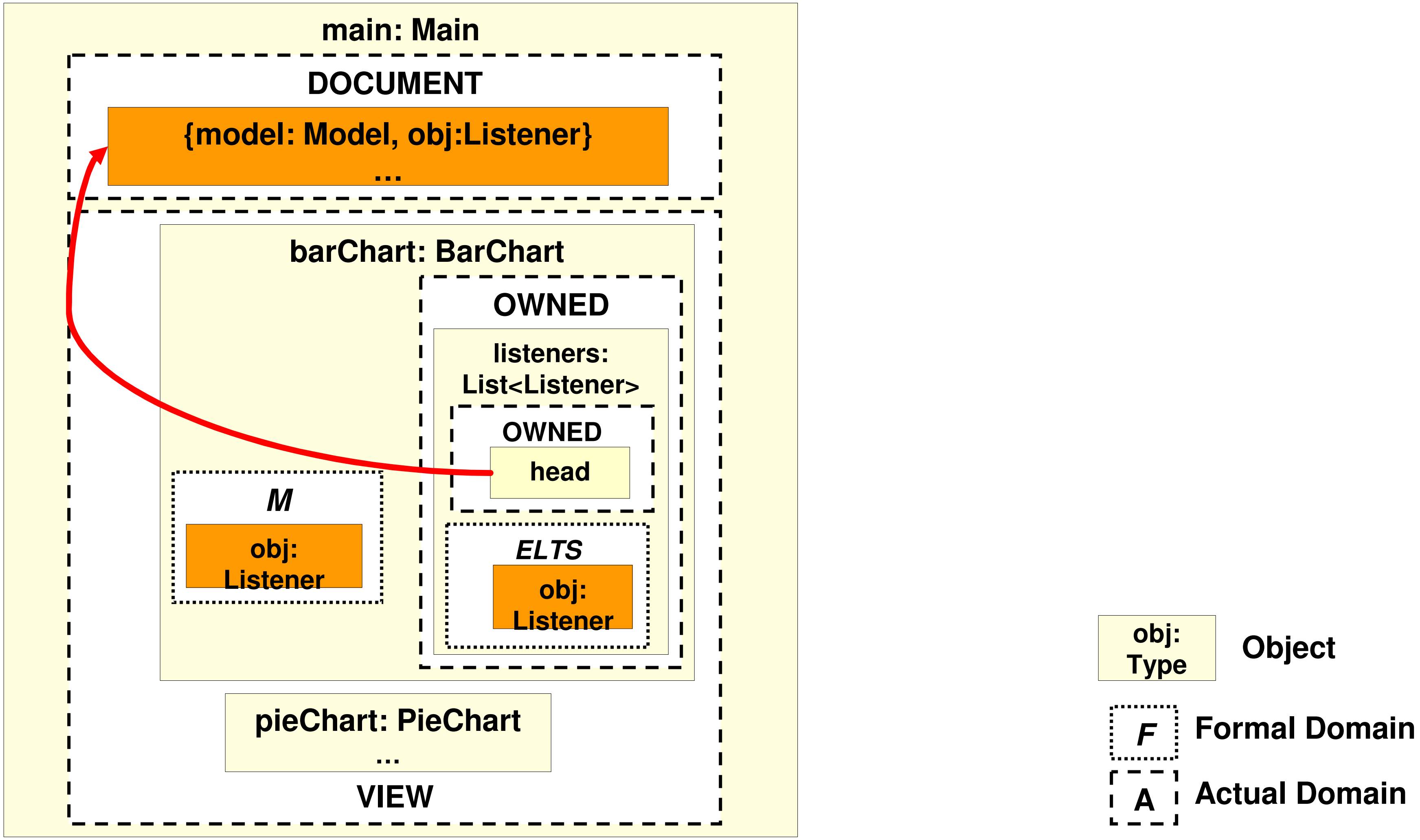
Aliasing precision:

- Two objects in different domains cannot alias
- Two objects in same domain *may* alias

```
class Model implements Listener {
…
}
```



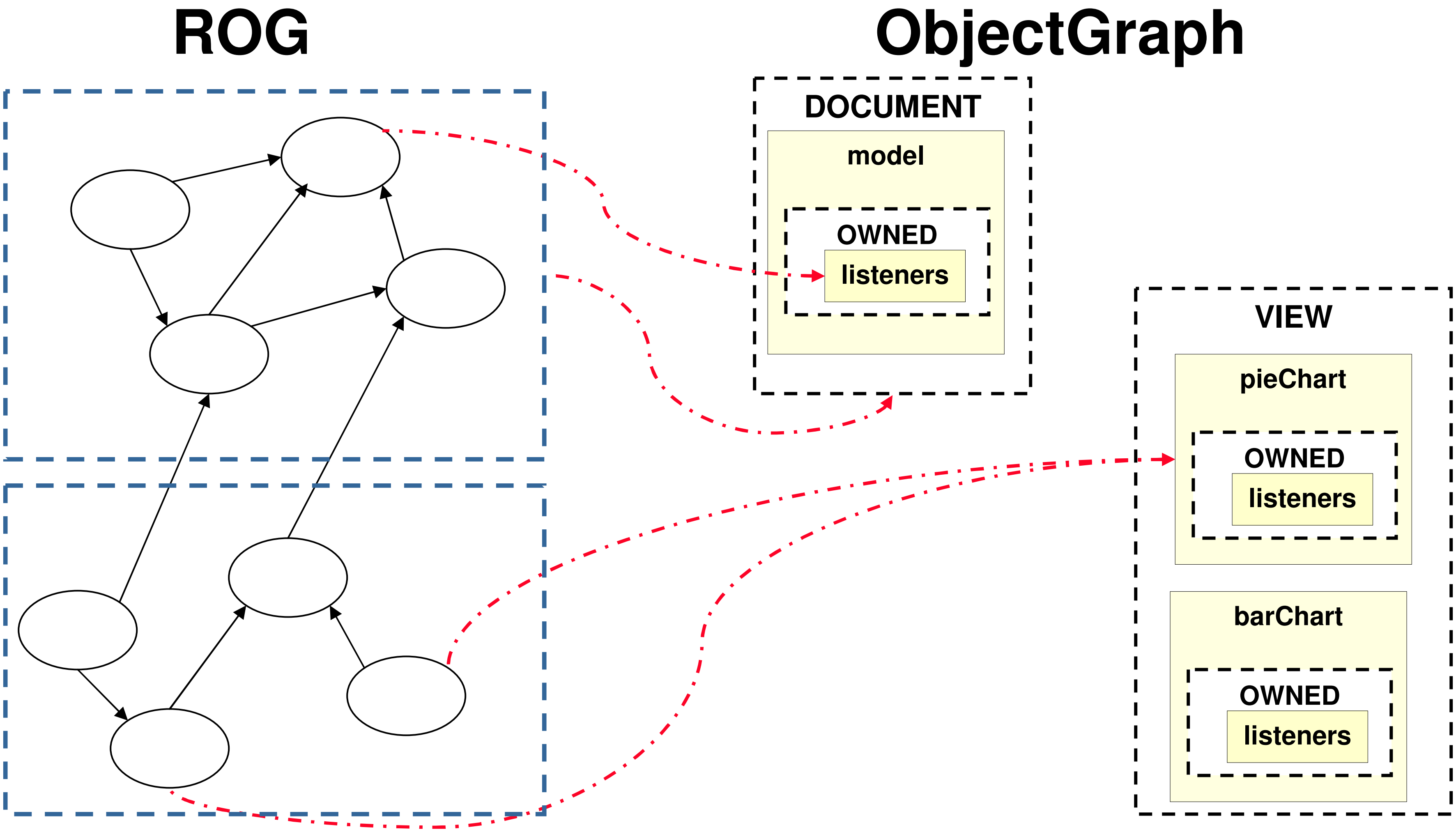
ObjectGraph: add edges to represent field references (points-to)



Soundness of extracted architectures

- Relate store typing to extracted **ObjectGraph**
  - Ownership domains type system
  - Featherweight Java formalization
- Every runtime object in any true Runtime Object Graph (ROG) maps to **exactly one** object in the ObjectGraph
- i.e., no one runtime object appears as two separate boxes in the diagram

Intuition behind soundness



Evaluation of static analysis

- Research Question: *does an extracted architecture suffer from too much or too abstraction?*
- Extended examples
  - JHotDraw (**15 KLOC**)
  - HillClimber (**15 KLOC**)
  - Aphyds (**8 KLOC**)
- Field Study
  - LbGrid (**30 KLOC**)

Why a field study?

- Generally accepted research method
- Evaluate how well tool or technique works with real code and users
- Evaluate adoptability claims

Field Study

- Research Questions
- Setup and Methodology
- Extraction Process
- Quantitative Data
- Qualitative Data

Research Questions

- Main questions:
  - **RQ #1:** Will outside developer **understand abstraction by ownership hierarchy**?
  - **RQ #2: How to annotate** a real system? How much effort will it take?
- Subsidiary questions:
  - **RQ #3:** How can we improve the tool's **usability** and identify **missing features**?
  - **RQ #4:** Can one add annotations for the **top-level architecture**, then extend those annotations down?

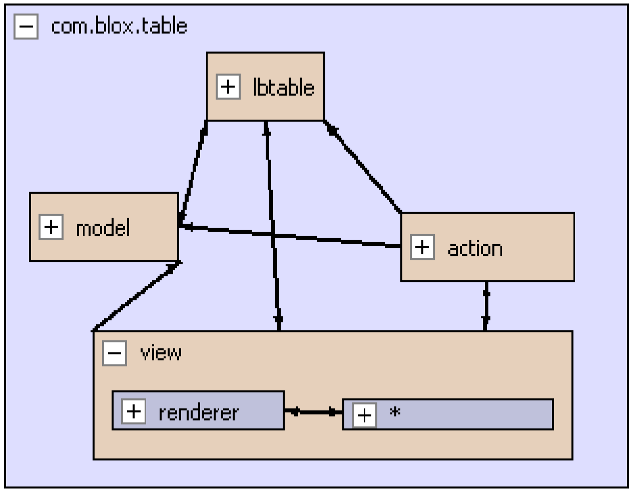
Setup and Methodology

- On-site week-long field study
- Select target portion of system
- **Communicate** with original developers to understand their architectural intent
- **Add annotations** and **typecheck** them
- **Extract** runtime architecture using **static analysis**
- **Show snapshots** to developer
- **Refine annotations** based on **feedback**
- **Address** annotation **warnings**

Subject System: LbGrid

- Grid control
- **30 KLOC**
- Part of **250-KLOC** commercial system
- **300 classes/interfaces**

  (even more when counting non-static inner classes)
- Developer familiar with code available at location (13 years of experience)

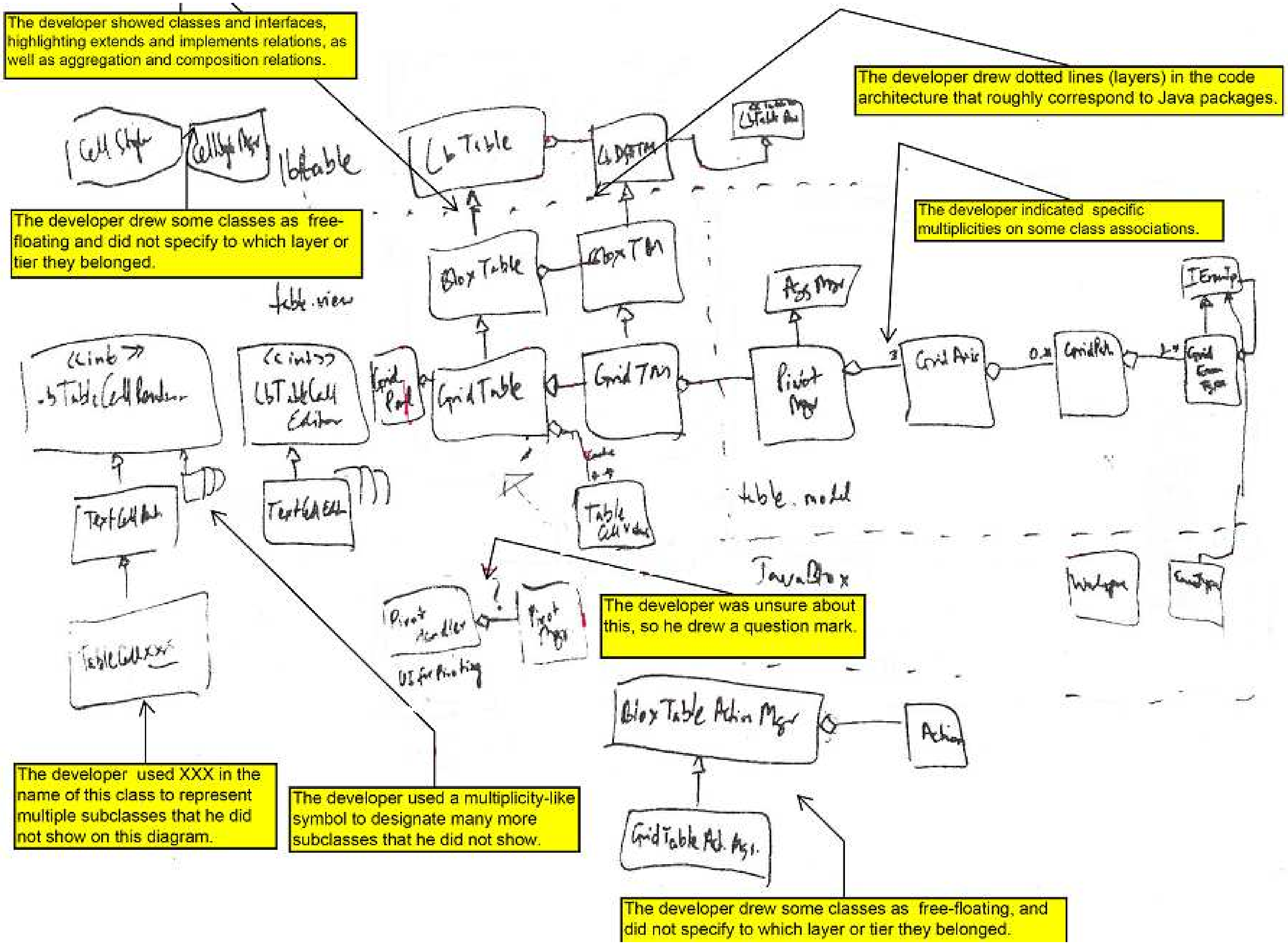


High-level LbGrid module view, obtained using Lattix LDM.
A box represents a package.

Getting started

- Ideally, get developer document as-designed **runtime architecture**
  - Could be existing documentation
  - Identify architectural decomposition
- Developer drew a code architecture!
  - Many existing tools could produce that
  - Created more abstracted version

Developer’s code architecture



The developer showed classes and interfaces, highlighting extends and implements relations, as well as aggregation and composition relations.

The developer drew dotted lines (layers) in the code architecture that roughly correspond to Java packages.

The developer drew some classes as free-floating and did not specify to which layer or tier they belonged.

The developer indicated specific multiplicities on some class associations.

The developer was unsure about this, so he drew a question mark.

The developer used XXX in the name of this class to represent multiple subclasses that he did not show on this diagram.

The developer used a multiplicity-like symbol to designate many more subclasses that he did not show.

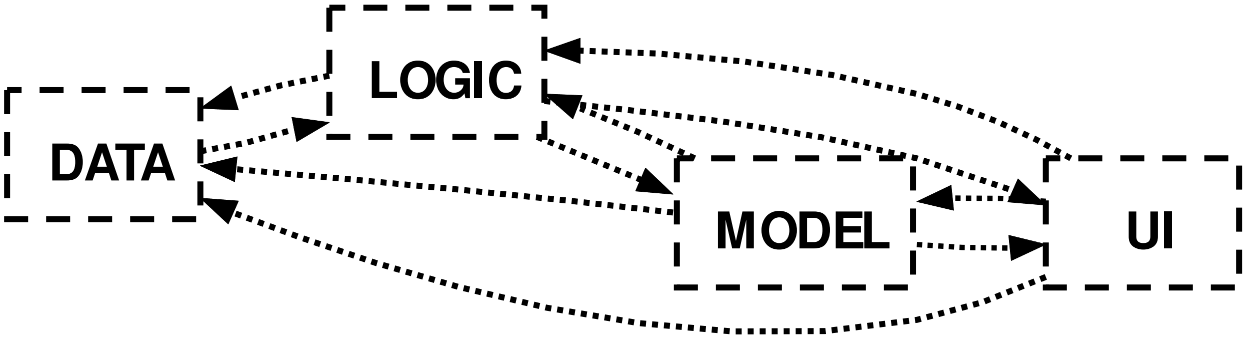
The developer drew some classes as free-floating, and did not specify to which layer or tier they belonged.

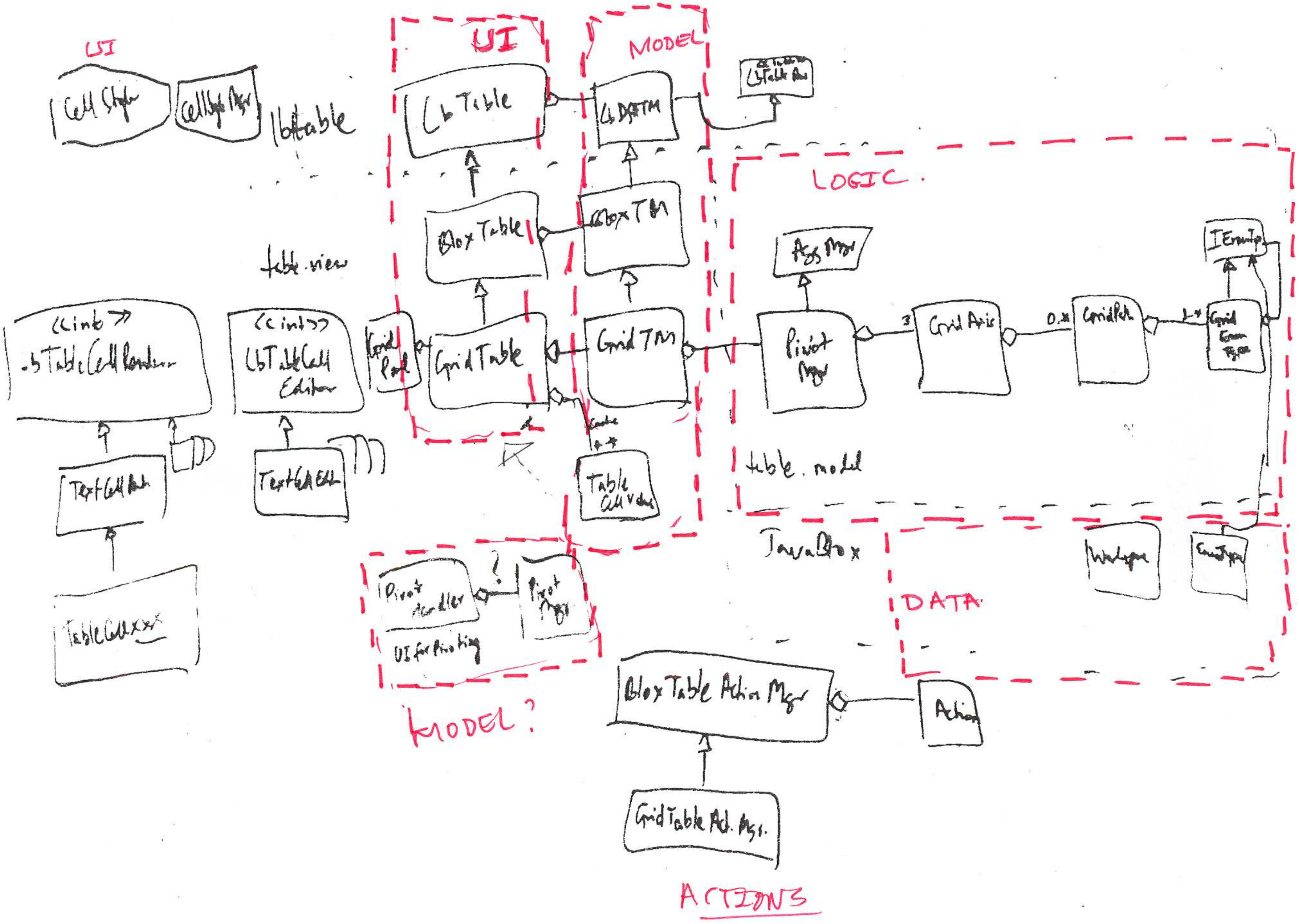
Architectural Extraction Process

1. Choose **top-level domains**
2. Map objects to domains
3. Achieve **desired number of objects** in each domain
4. Achieve appropriate **visual detail**
5. Address **annotation warnings**

1. Choose the top-level domains

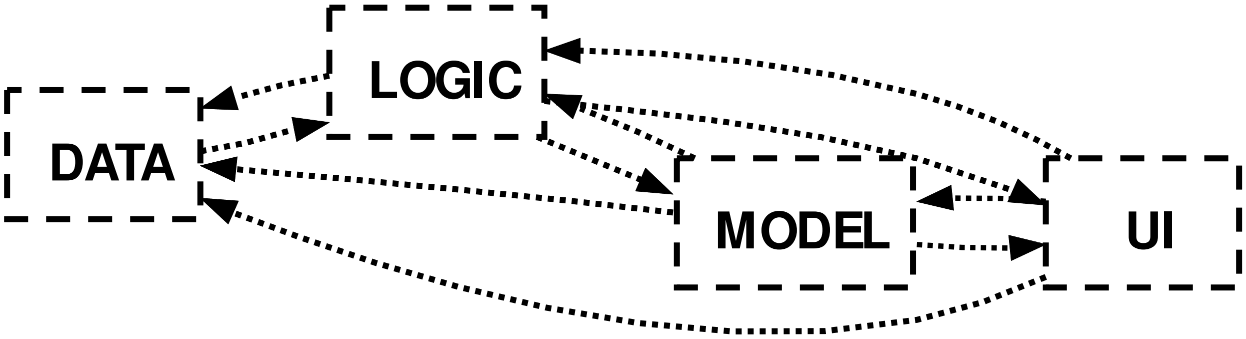
- **UI:** user interface objects
- **MODEL:** model for user interface objects
- **LOGIC:** business logic objects
- **DATA:** data access objects




UI
Cell Style
CellStyleMgr
lbtable
UI
MODEL
LbTable
LbDGTM
LbTableRow
LOGIC
BloxTable
BloxTM
AggMgr
IEnumType
table.view
<<int>>
LbTableCellRenderer
<<int>>
LbTableCell Editor
Grid Pad
GridTable
GridTM
Pivot Mgr
3
Grid Axis
0..*
GridPath
1..*
Grid Enum Type
cache
0..*
Table Cell Value
TextCellRdr
TextCellEdit
table.model
TableCellxxx
JavaBlox
WeekType
EnumType
Pivot Handler
?
Pivot Mgr
DATA
UI for Pivoting
BloxTable Action Mgr
Action
MODEL?
GridTable Act. Mgr.
ACTIONS

2. Map objects to domains

- Start with top-level domains
  - **UI:** instances of LbTable, etc.
  - **MODEL:** instances of LbTableModel, etc.
  - **LOGIC:** instances of PivotManager, etc.
  - **DATA:** instances of Workspace, Predicate, etc.



3. Achieve desired number of objects in each domain

- Push **secondary objects under** primary objects
- Use **abstraction by types** to merge objects

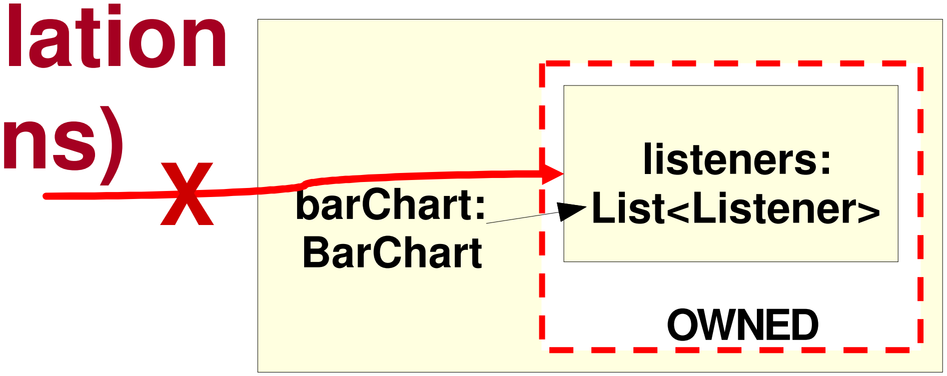
Questions to the developer

- ***"Is this instance of type T in tier D?"***
  - Help **map objects to domains**
  - As first approximation, map types to domains
  - Two instances of the same type, e.g., ArrayList, could map to different domains
- **"Is component *X* in tier D conceptually part of component Y, so I can push X under Y?"**

Abstraction by Ownership Hierarchy

- Push **secondary object** under **primary object** using

(1) Strict encapsulation (private domains)



Using **Strict Encapsulation**

- Push secondary object into **private domain** of primary object

```
class RequestBuilder {
    private List predicates;

    public List getPredicates() {
       return this.predicates;
    }
  }
```

Using Strict Encapsulation

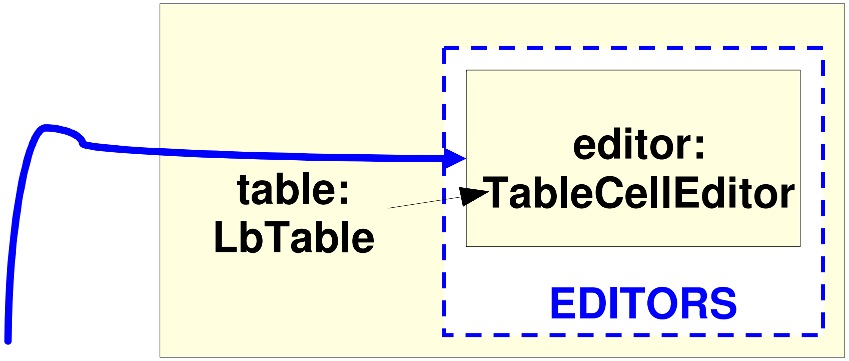
- Sometimes **change code** to **return copy** of list instead of alias

```
   class RequestBuilder {
     domain OWNED;
     OWNED List predicates;

   public UNIQUE List getPredicates() {
     UNIQUE List copy = new List();
     ...
     return copy;
   }
}
```

Abstraction by Ownership Hierarchy

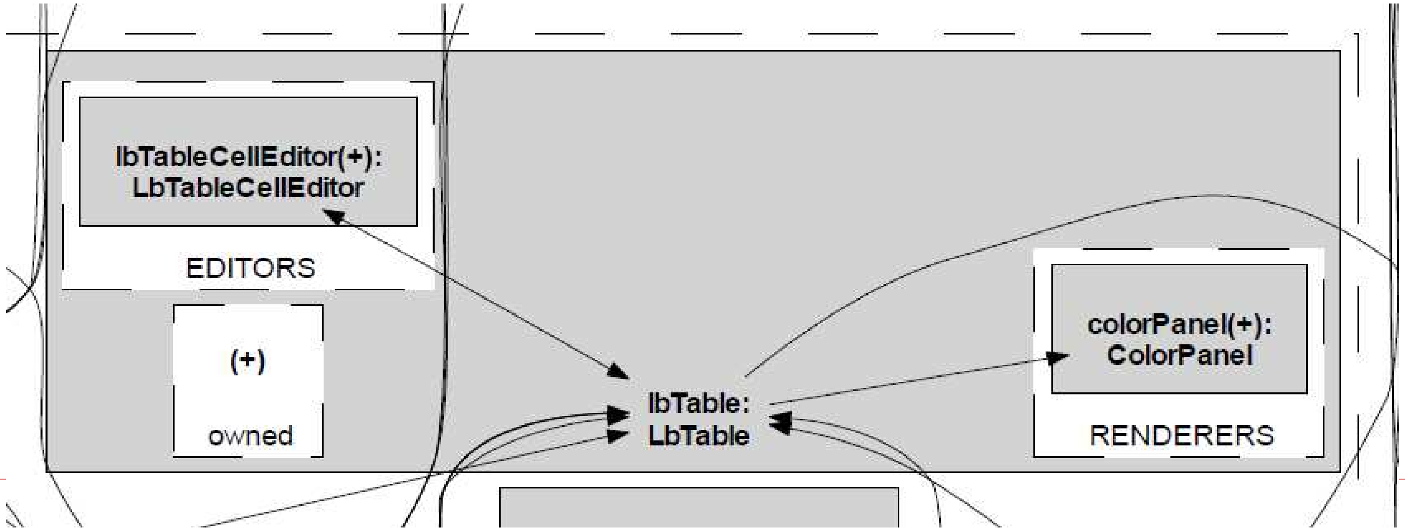
- Push **secondary object under** primary object using

(2) Logical containment (public domains)



Using **Logical Containment**

- Push secondary object into **public domain** of primary object
  - **LbTable.RENDERERS**: TextCellRenderer, ColorCellRenderer, etc.
  - **LbTable. EDITORS**: TextCellEditor, ColorCellEditor, etc.



4. Achieve appropriate **visual detail**

- **Collapse** or **expand sub-structure** of objects
    - Select core objects
    - E.g., expand public domains
    - E.g., collapse private domains

- Change **projection depth** across all objects

Quantitative Data

- **35 hours** to annotate 30 KLOC
  - 30 hours on-site
  - 5 hours off-site
- **4,000** remaining warnings

Qualitative Data

- The developer understood
  - assigning components to runtime tiers
  - **abstraction by ownership hierarchy**
  - object abstraction (merging)

The developer understood abstraction by ownership hierarchy

- Developer identified objects that must be pushed underneath others:

  “The following are **too low-level to be at the outermost tier**: CellPosition, ...”

  Developer M.

- Developer wished to **examine an object’s sub-structure**
  - We provided him with hard-copy snapshots
  - We implemented standalone viewer since

Qualitative Data

- The developer seemed confused by:
  - lack of multiplicities
  - object labeling
  - **level of detail**
- E.g., developer’s code architecture
  - Heavily abstracted
  - << 300 types

Principled vs. unprincipled approach

- Principled approach: push **secondary objects under** primary objects
  - Change annotations
  - Optionally change code
  - Can get tedious
- Unprincipled approach: **select and elide any object or domain** in extracted architecture
- Soundness argument:
  - Should architecture reflect everything?
  - Or only objects of interest to developer?

Future Work

- Produce task-specific views
- Better tools to add annotations
- **Check conformance** of as-built to as-designed architecture
    - Requires developer to draw as-designed runtime architecture
- Study benefits of runtime architectures:
    - Identify **code modification tasks** where runtime architecture crucial

Related Work

- Dynamic analyses
  - Or mix of static and dynamic analyses
  - Usual coverage issues with dynamic analyses
- Static analyses
  - With or without annotations
  - Non-hierarchical object graphs
- Library-based solutions
  - Re-implement on architectural middleware
- Language-based solutions
  - Re-engineer system to extended language

Adoptability of annotation-based approach

- Previously studied language-based solutions
  - Re-engineering to **ArchJava** (Aldrich et al.,ICSE'02)
  - Specify in code **component classes**, ports
  - Imposes implementation restrictions, e.g., **cannot return reference** to instance of component class
- **Annotation-based approach more adoptable** than re-engineering to ArchJava-like languages
  - Could not have re-engineered to ArchJava in 35 hrs
  - Even after accounting for tool familiarity

Conclusion

- **Static analysis compelling**
  - Difficult to setup and run system
  - No need to learn how to use system
- Developer **understood abstraction by ownership hierarchy**
- **Annotation-based approach more adoptable** than re-engineering