Static Extraction of Hierarchical Runtime Object Graphs – Tool Demonstration

Marwan Abi-Antoun   Jonathan Aldrich

School of Computer Science

Carnegie Mellon University

OOPSLA
CONFERENCE

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)

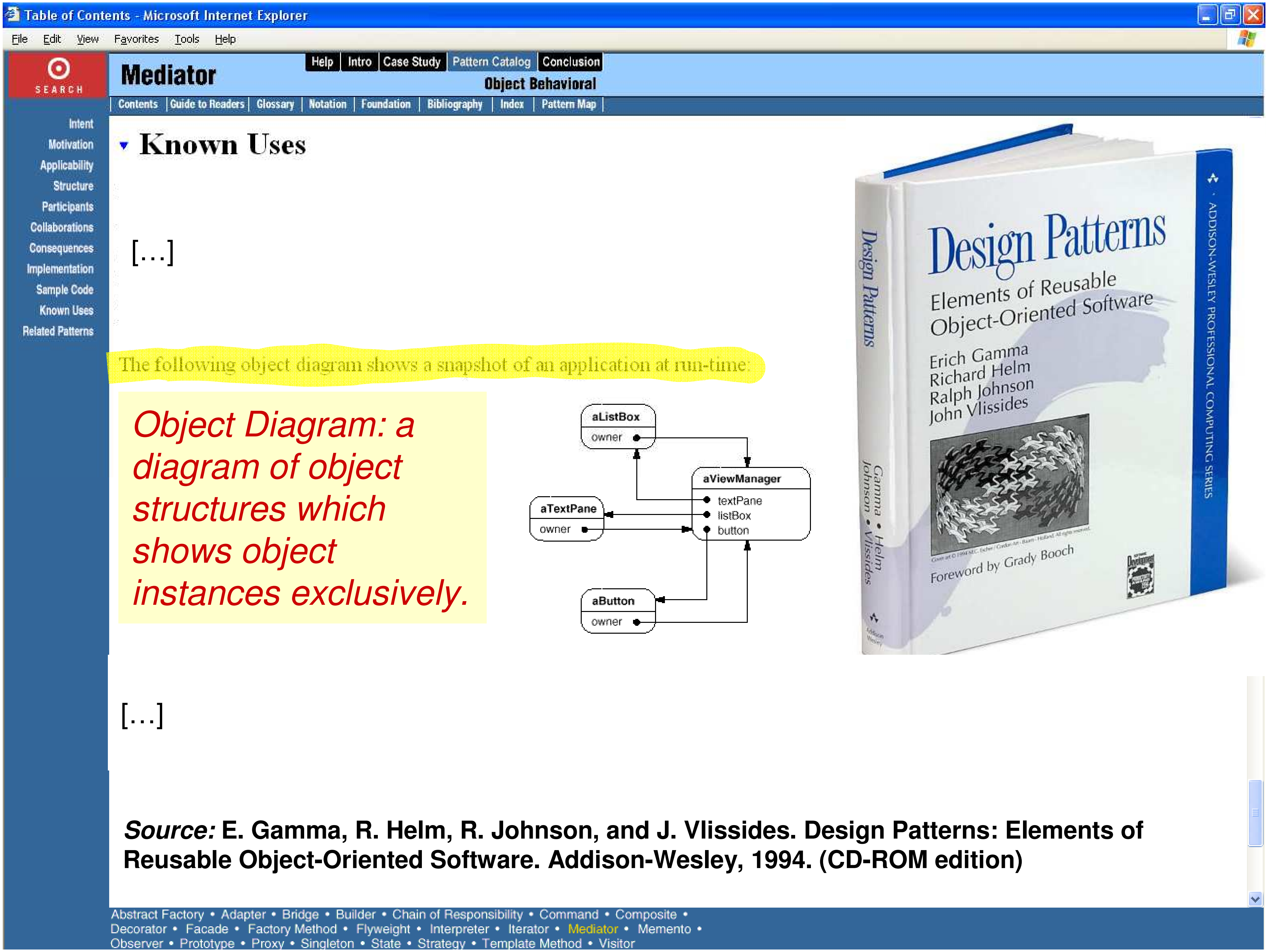


Table of Contents - Microsoft Internet Explorer
File
Edit
View
Favorites
Tools
Help
SEARCH
Help
Intro
Case Study
Pattern Catalog
Conclusion
Contents
Guide to Readers
Glossary
Notation
Foundation
Bibliography
Index
Pattern Map
Intent
Motivation
Applicability
Structure
Participants
Collaborations
Consequences
Implementation
Sample Code
Known Uses
Related Patterns

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.

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
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES
Gamma • Helm • Johnson • Vlissides

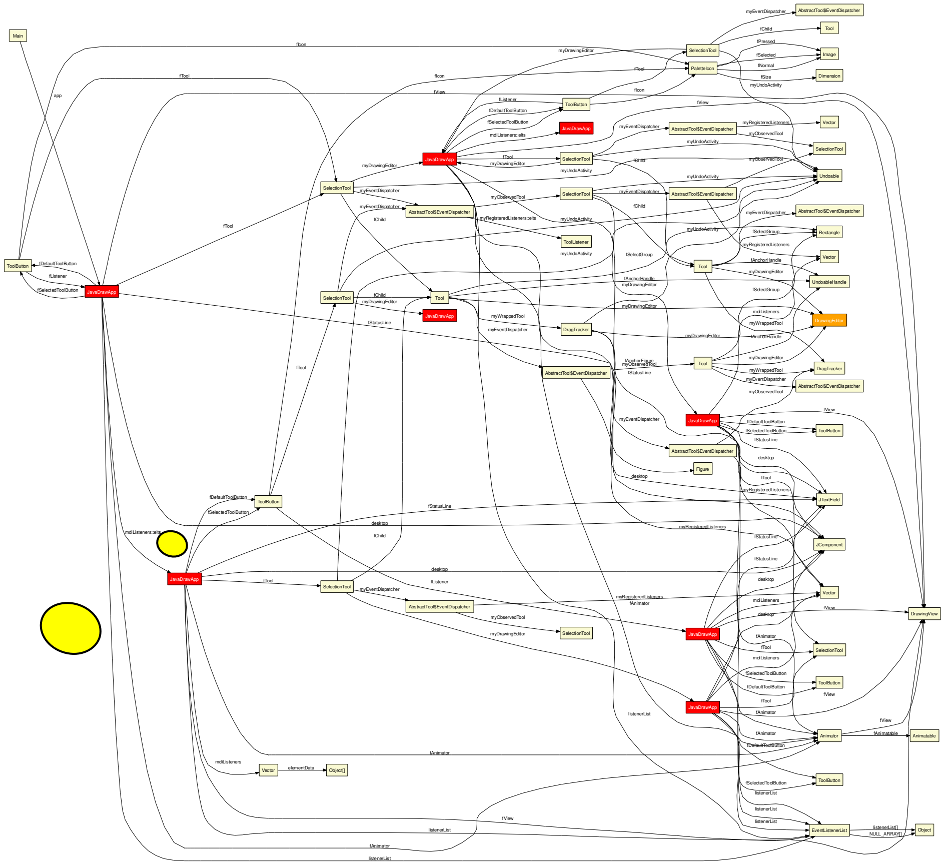
[…]

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

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

Tool support to extract runtime structure less mature

- Low-level objects
- No architectural abstraction
- Some analyses **incorrectly** handle **aliasing**



Output of Womble on JHotDraw (15 KLOC)

JavaDrawApp, DrawingEditor, represent one runtime object.

Key Insight

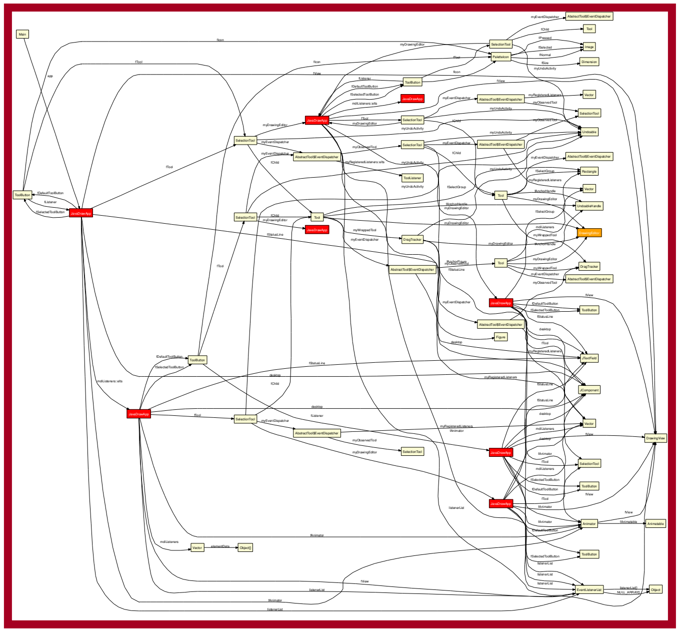
Ownership domain annotations enable the extraction of **sound hierarchical** object graphs using **static analysis**.

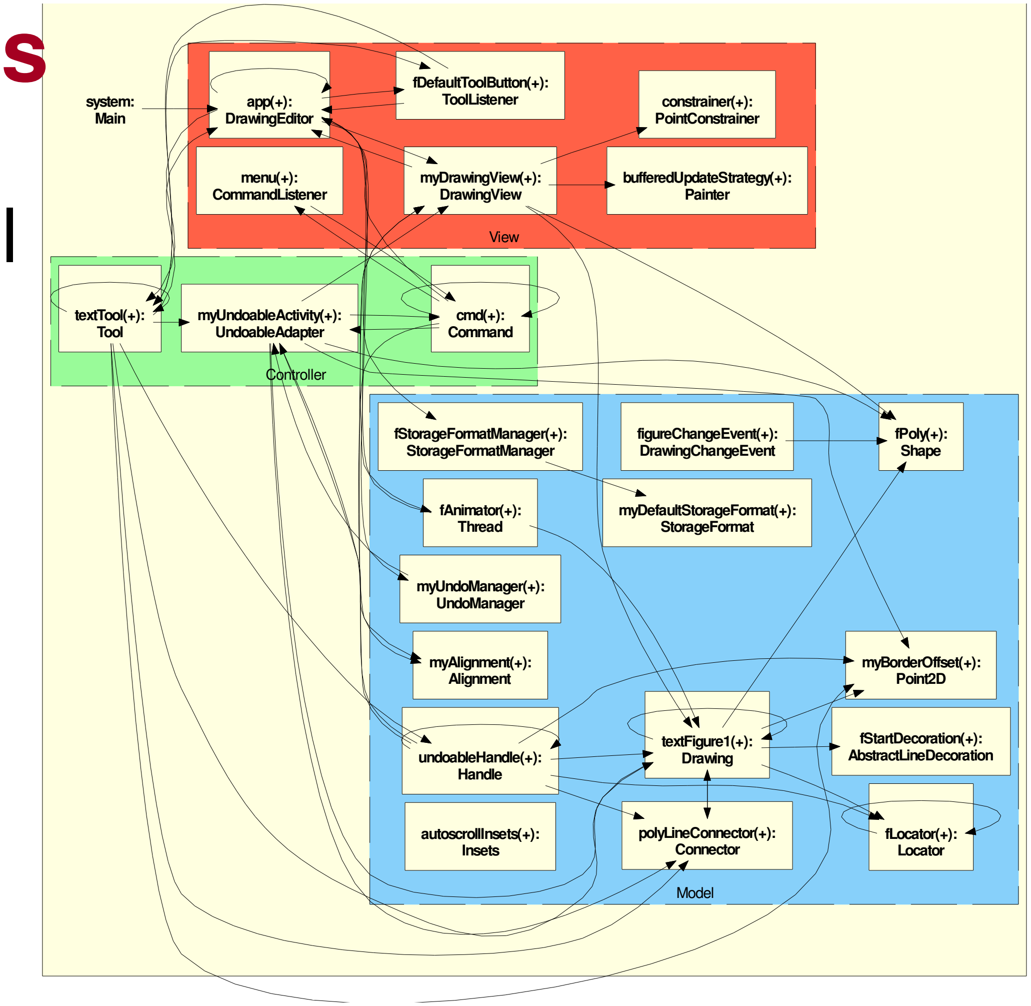
Extracting sound hierarchical object graphs using static analysis

- Why **static analysis**?
  - Dynamic analysis shows object graphs for a few program runs, not all
- Why **sound**?
  - To be most useful, show all objects and relations that could exist at runtime

Extracting sound hierarchical object graphs using static analysis

- **Hierarchical graphs**
  - Flat graphs do not provide architectural abstraction or scale




system: Main
app(+): DrawingEditor
fDefaultToolButton(+): ToolListener
constrainer(+): PointConstrainer
menu(+): CommandListener
myDrawingView(+): DrawingView
bufferedUpdateStrategy(+): Painter
View
textTool(+): Tool
myUndoableActivity(+): UndoableAdapter
cmd(+): Command
Controller
fStorageFormatManager(+): StorageFormatManager
figureChangeEvent(+): DrawingChangeEvent
fPoly(+): Shape
fAnimator(+): Thread
myDefaultStorageFormat(+): StorageFormat
myUndoManager(+): UndoManager
myAlignment(+): Alignment
myBorderOffset(+): Point2D
undoableHandle(+): Handle
textFigure1(+): Drawing
fStartDecoration(+): AbstractLineDecoration
autoscrollInsets(+): Insets
polyLineConnector(+): Connector
fLocator(+): Locator
Model

Demonstration Outline

- Ownership annotations
  - **Adding annotations**
  - **Typechecking** annotations
- Runtime structure
  - **Extraction tool**
- Real-World Example
  - JHotDraw
- Additional material
  - Static analysis

Ownership Domains

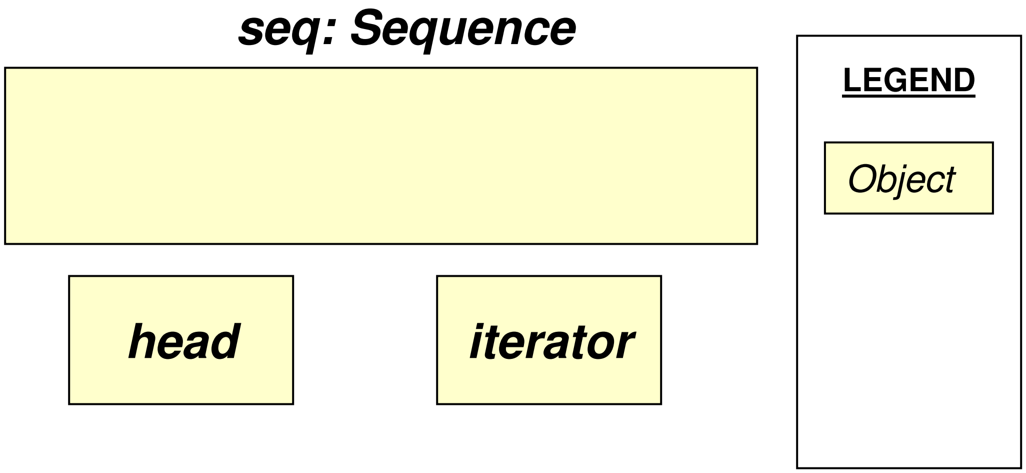
Ownership domains

[Aldrich and Chambers, ECOOP'04]

- Each object defines conceptual groups (*ownership domains)* to hold its state
- Separate object's internals from object's boundary (accessible to outside)
- Ensure private state not leaked
- Distinguish different "subsystems" within object

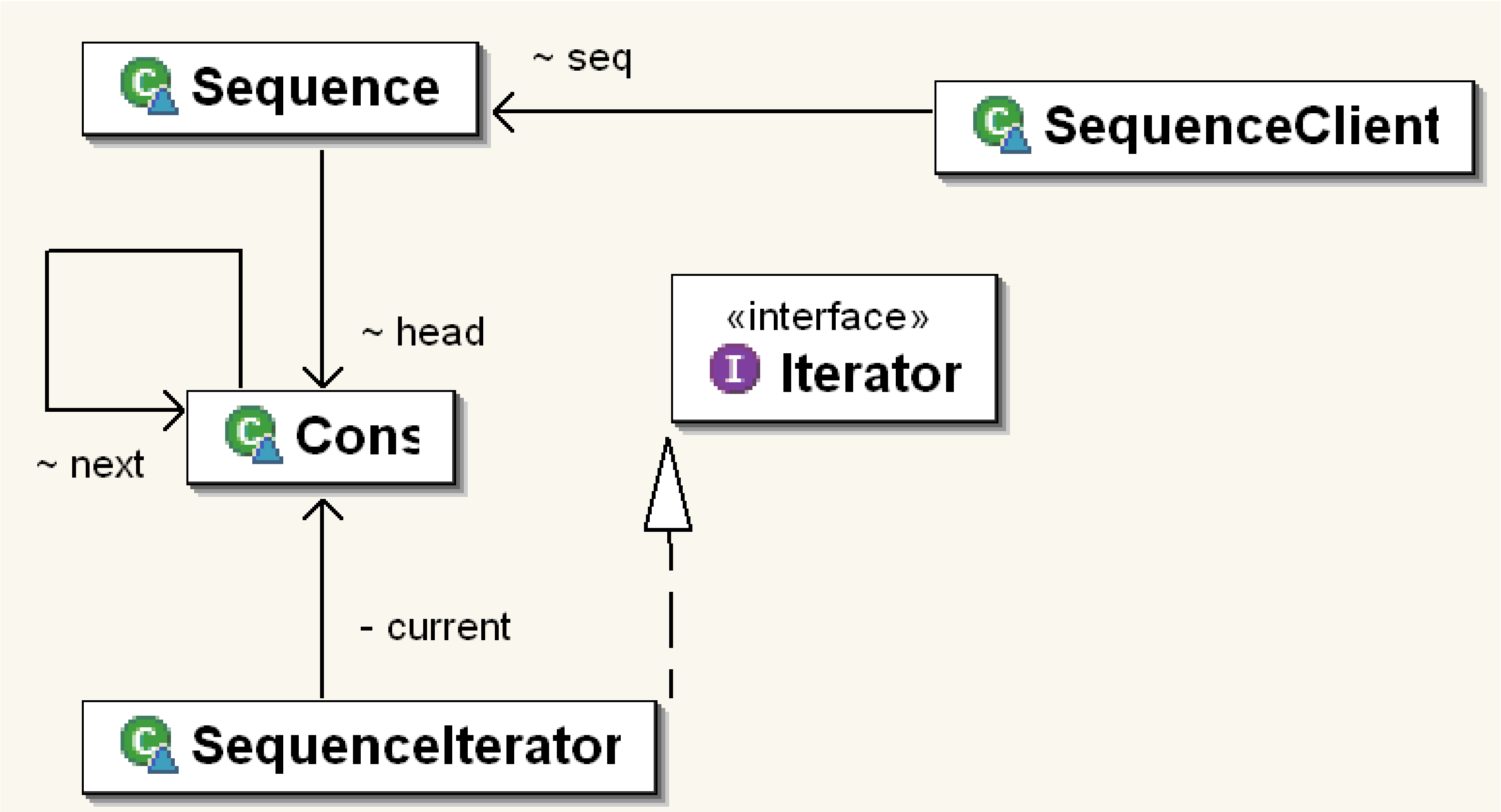
Example: Sequence

```
class Sequence {
                 Cons head;

 public
 Iterator iterator() {
    return new Iterator(head);
  }
}
```



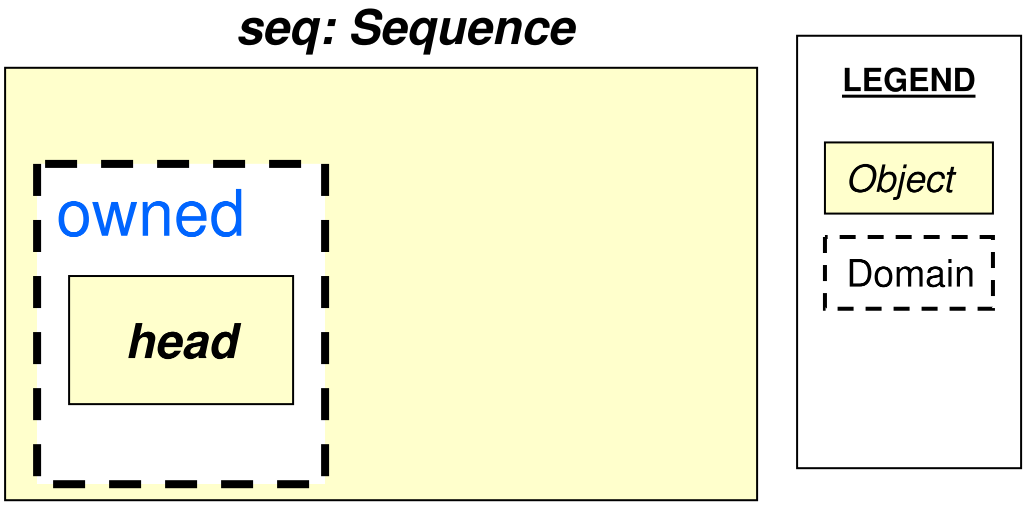
- Sequence has private state (head)
  - Should not be accessible to outside
- Sequence has iterators that are accessible to outside
  - Can also access private state

Sequence Code Structure



Sequence: Private Domain

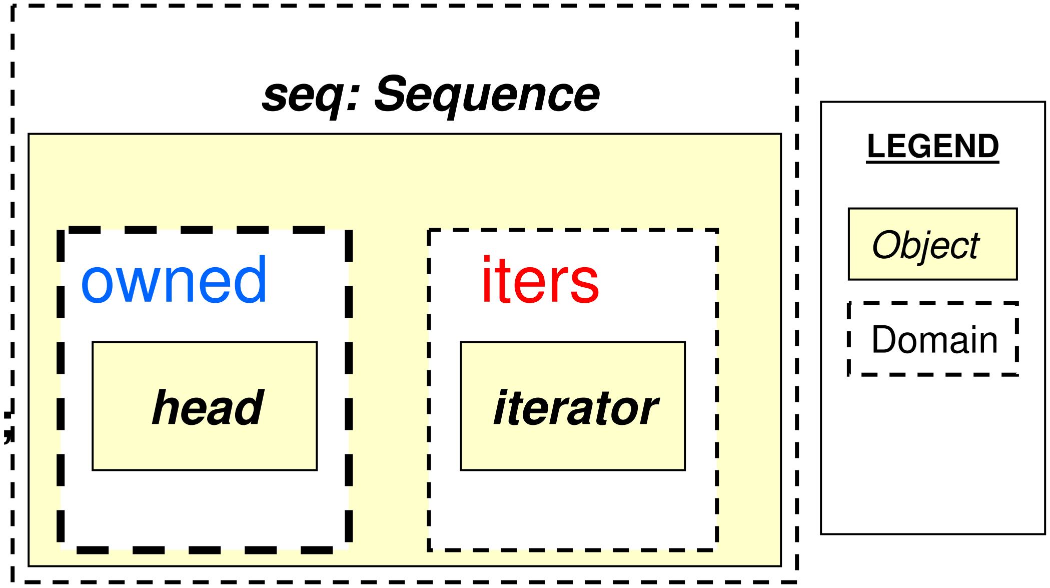
```
@Domains({“owned          })
class Sequence {
 @Domain(“owned”) Cons head;

 public
 Iterator iterator() {
    return new Iterator(head);
  }
}
```



- Each object has one or more domains
  - E.g., Sequence declares domains owned and iters
- Each object is in exactly one domain
  - E.g., head in domain owned; iterator in domain iters

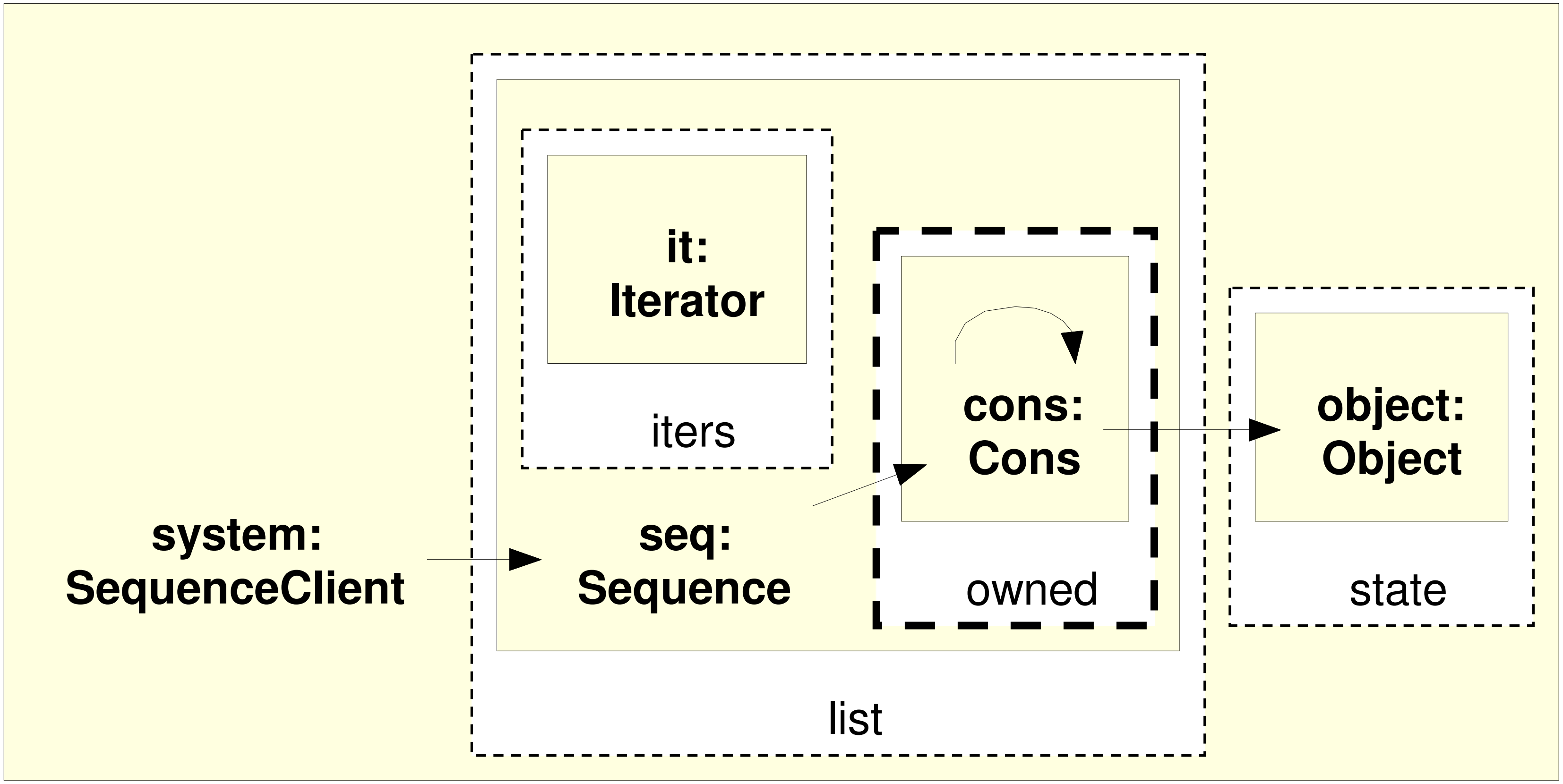
Sequence: Public Domain

```
@Domains({“owned”, “iters”})
class Sequence {
 @Domain(“owned”) Cons head;

 public @Domain(“iters”)
 Iterator iterator() {
    return new Iterator(head);
  }
}
```



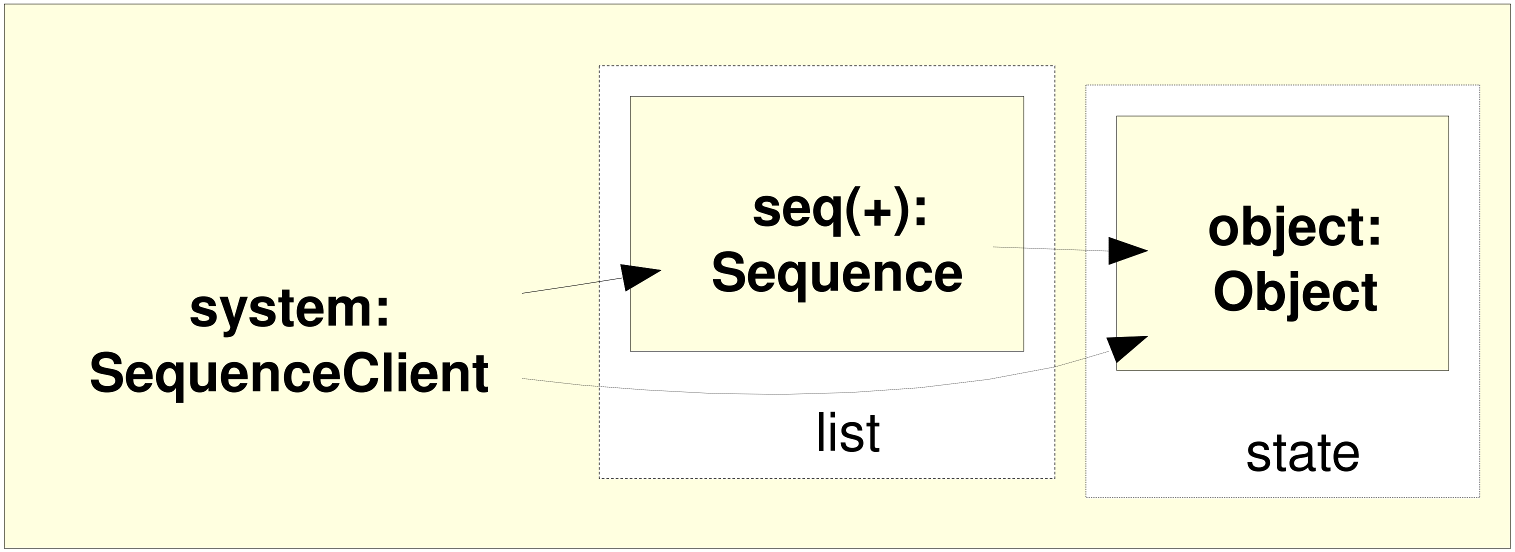
- Each object has one or more domains
  - E.g., Sequence declares domains owned and iters
- Each object is in exactly one domain
  - E.g., head in domain owned; iterator in domain iters

Sequence Runtime Structure



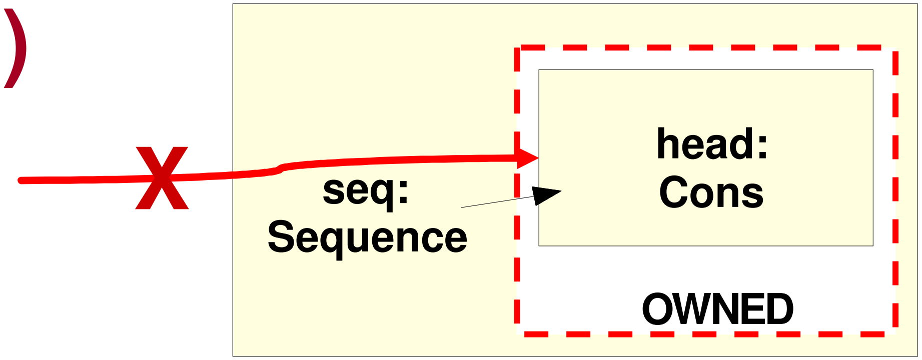
Sequence Runtime Structure

- Collapse Sequence’s sub-structure

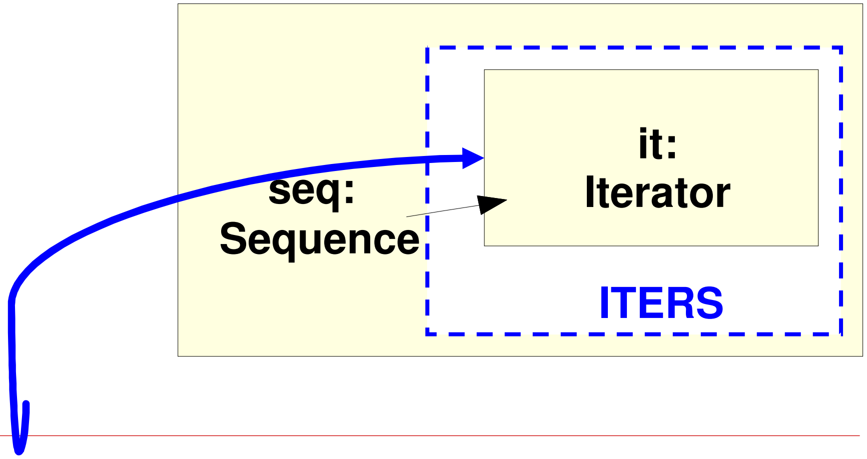


Encapsulation and Containment

(1) Strict encapsulation (private domain)



(2) Logical containment (public domain)



Annotation Tool Support

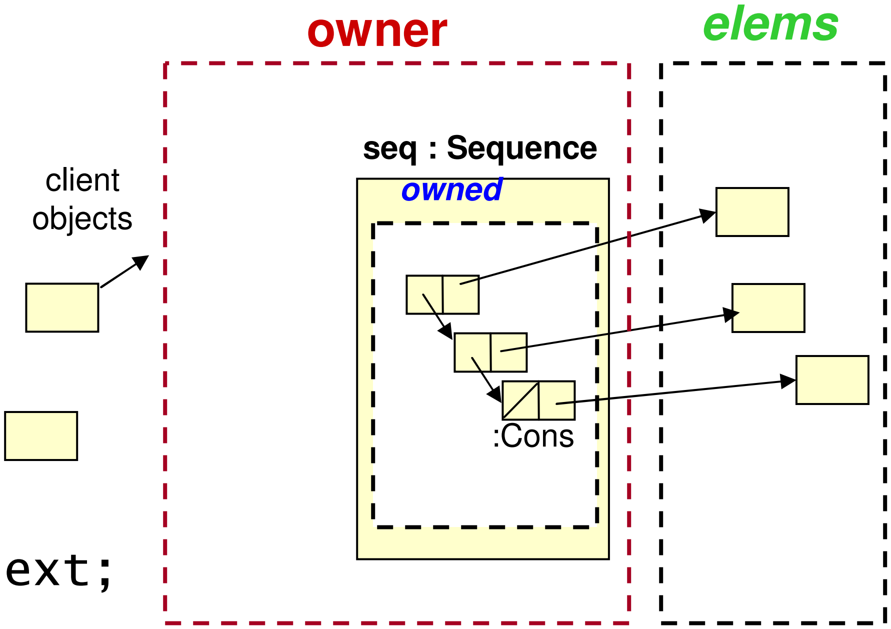
- Use **Java 1.5 annotations**
- Typechecker uses Eclipse JDT
- Warnings in Eclipse’s **problem window**

Demo: Checking Sequence

- Cannot return head of list
  - Head of list in **private domain**
  - **Stronger than making field private**
- Cannot nullify head of list
  - Stronger than Java visibility (e.g., **private**)
- Iterate over list
  - Iterator in **public domain**

Ownership Domain Parameters

```
@DomainParams({“elems”})
class Sequence {
 @Domain(“owned<elems>”)
 Cons head;
…
}
@DomainParams({“elems”})
class Cons {
  @Domain(“elems”) Object obj;
  @Domain(“owner<elems>”) Cons next;
}
```



- To share objects across domains
- Add domain parameter to hold elements in list
- **Implicit domain parameter “owner”**
  (Same as me, a.k.a. “peer” or “same”)

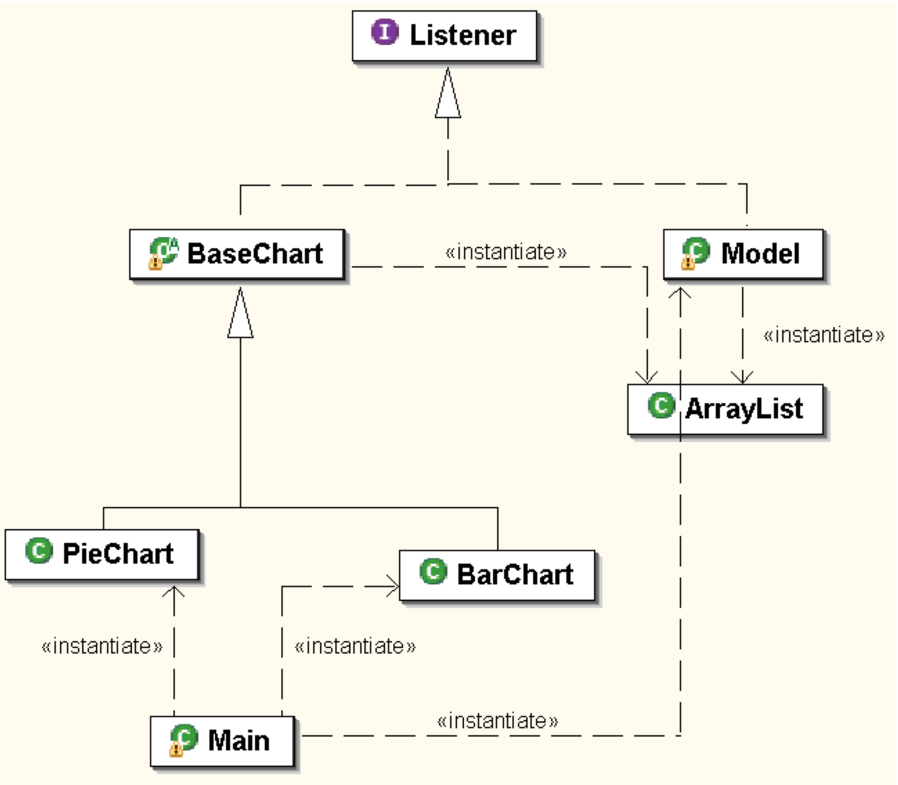
Demo: Annotating Listeners (Iteration 1)

Listeners Example

- Listeners tricky in object-oriented code
- Reuse annotated Sequence
  - Disguised as ArrayList

Listeners Code Structure

```
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 by Eclipse UML.

Demo: Listeners example

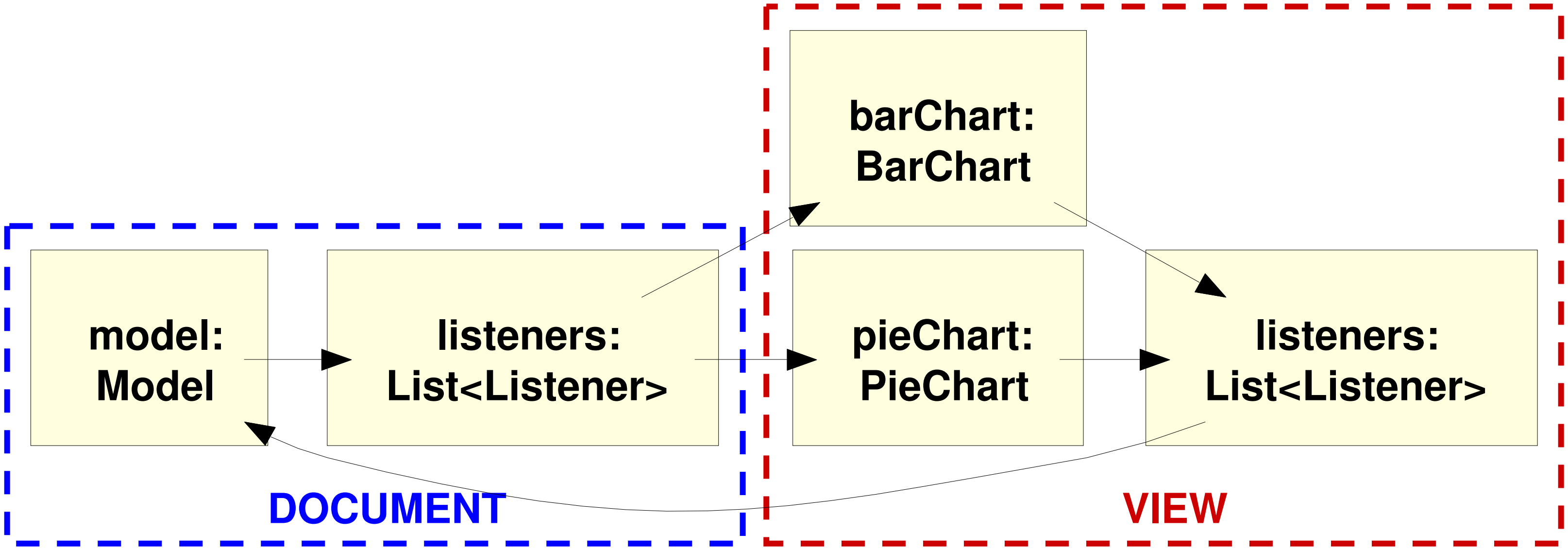
- Tool to add default annotations
  - Declare **owned** private domain
  - Private field place in domain **owned**
    - **owned:** object fully encapsulated
  - String mark **shared**
    - **shared:** shared persistently or globally
  - Method parameter mark **lent**
    - **lent:** temporary alias within method
- Not a smart inference tool!

Standard and third-party libraries

- Annotate external code
  - Ideally, library provider adds annotations
  - Annotations shared amongst authors
- Only annotate parts of library in use
- Wizard to generate skeleton XML file

Listeners Runtime Structure (version 1)

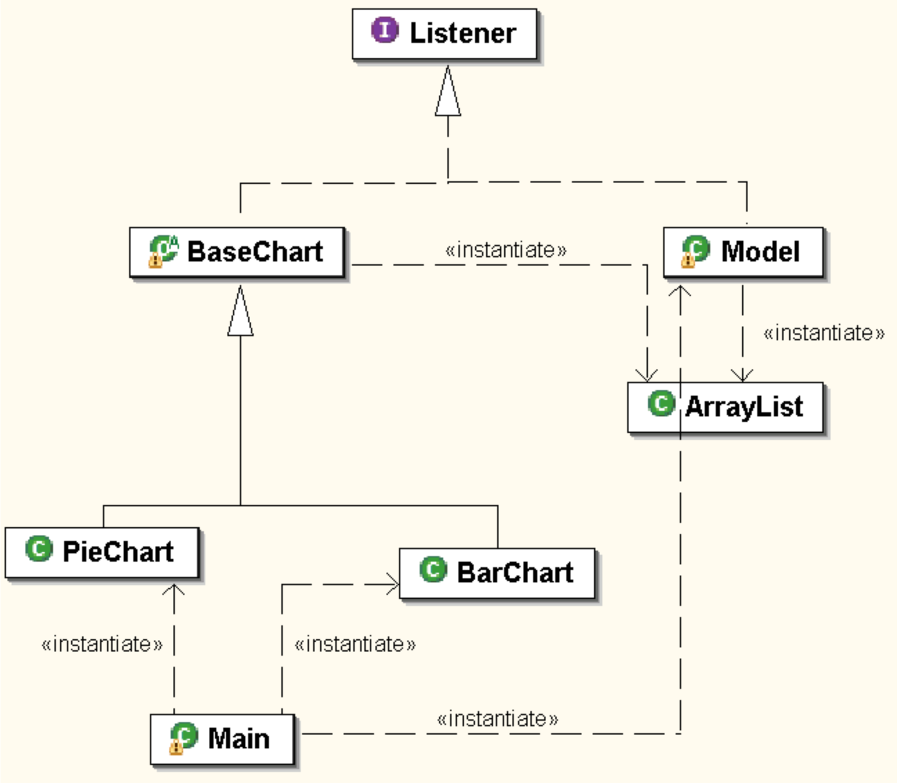
- Listeners at the top-level



Runtime Structure

Code Structure – Take 1

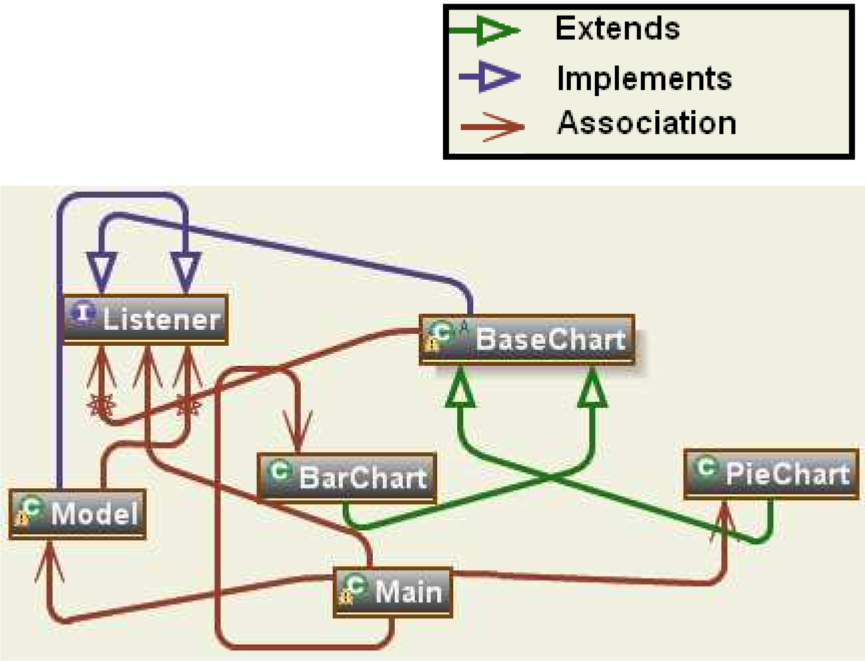
```
interface Listener { }

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 extracted by Eclipse UML.

Code Structure – Take 2

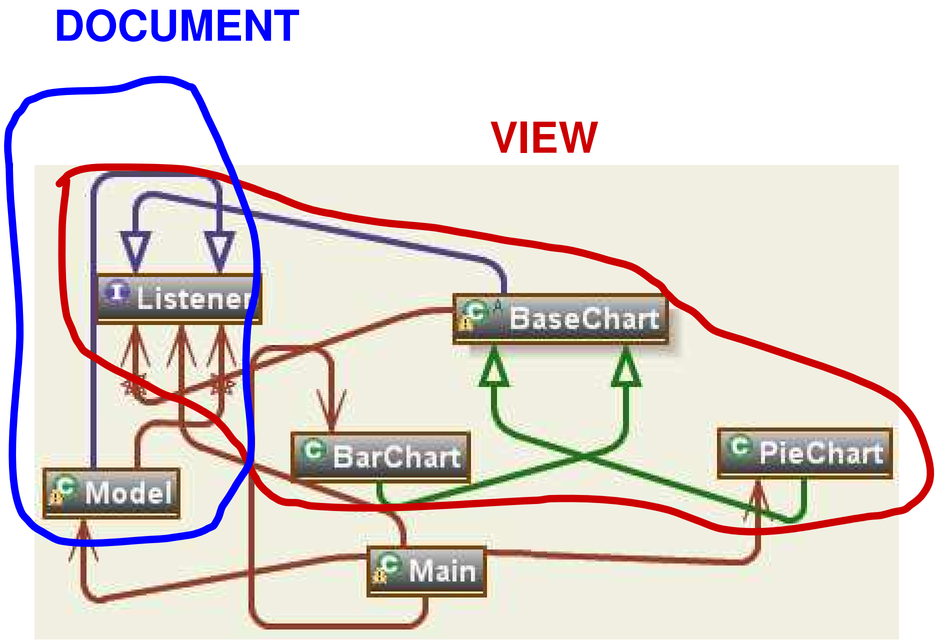
```
interface Listener { }

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 by AgileJ.

Code vs. Runtime Structure

- Who points to who?
- Do not distinguish between conceptually different instances of same class
- Extra details: abstract classes, interfaces, etc.
- No hierarchy



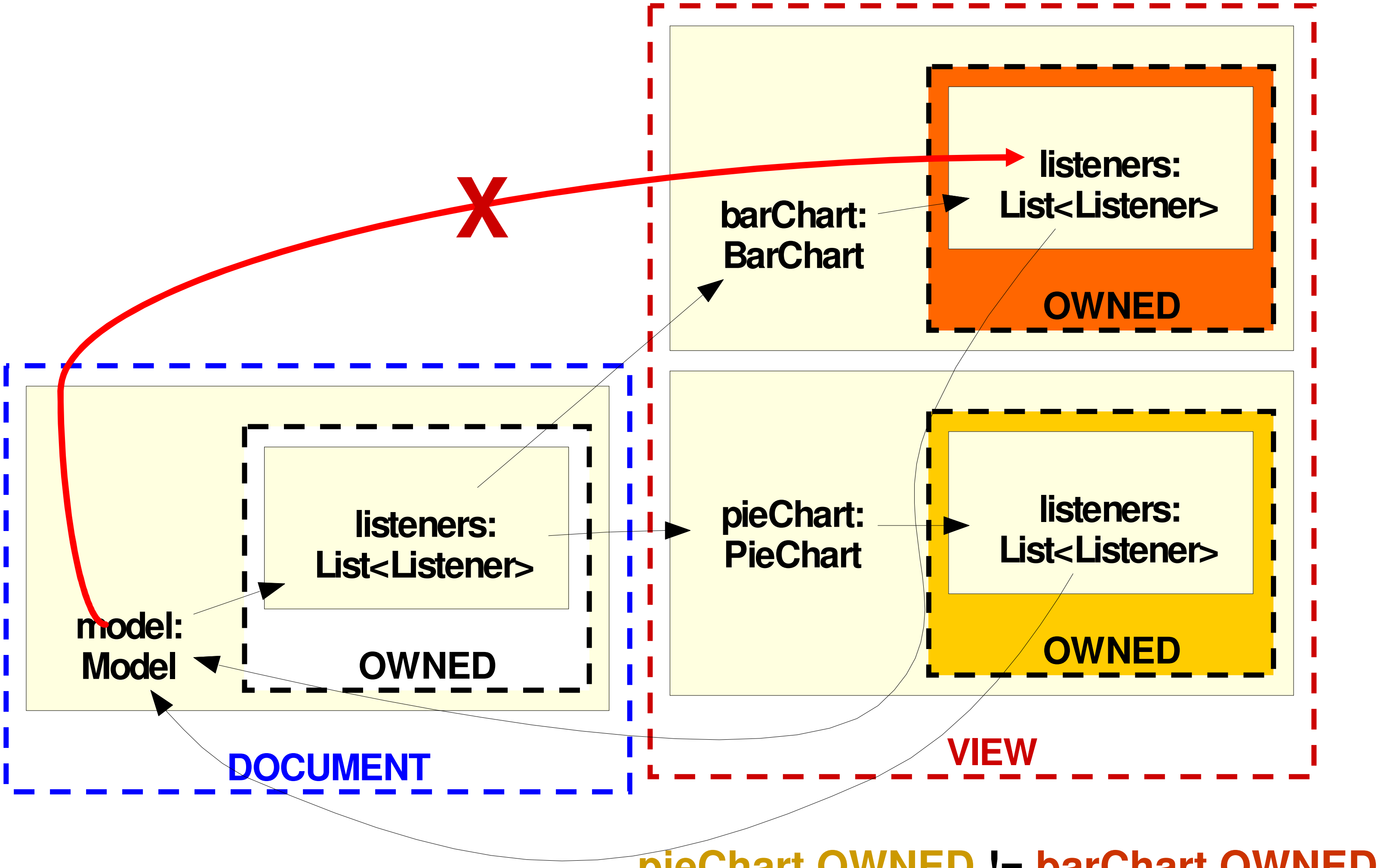
Class diagram extracted by AgileJ.

Demo: Annotating Listeners (Iteration 2)

Change annotations

- Instance encapsulation
- May require changing code to avoid *representation exposure, e.g.,*
    - Return copy instead of alias to internal List
    - Pass object linearly

Listeners Runtime Structure (version 2)

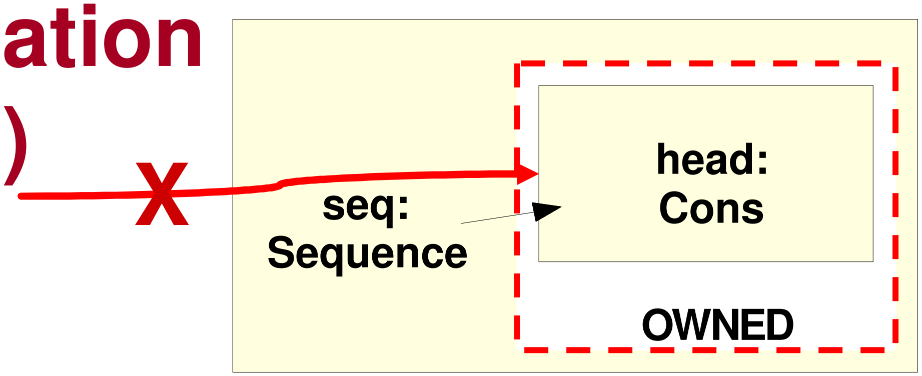


pieChart.OWNED != barChart.OWNED

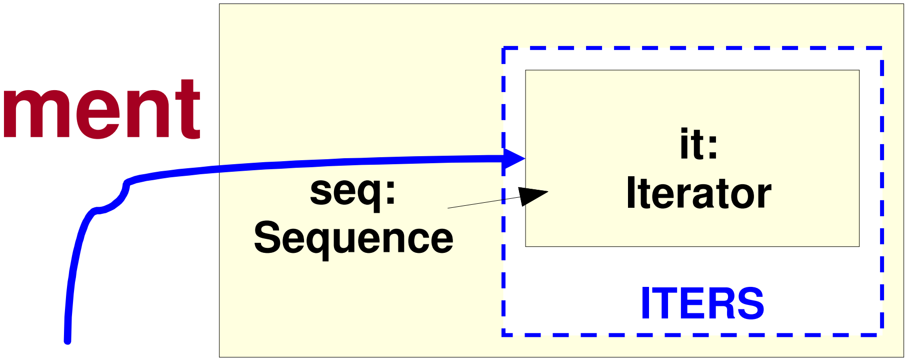
Abstraction by Ownership Hierarchy

- Push **secondary** objects **under primary** objects using

(1) Strict encapsulation (private domain**)**

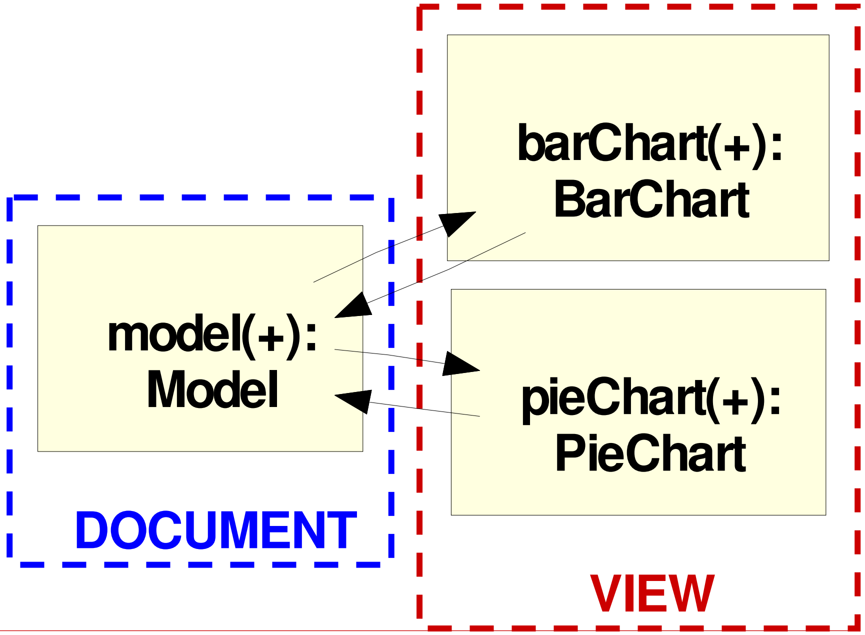


(2) Logical containment (public domain**)**



Hierarchy Provides Abstraction

- Can collapse object sub-structure
- Summary edges account for hidden objects

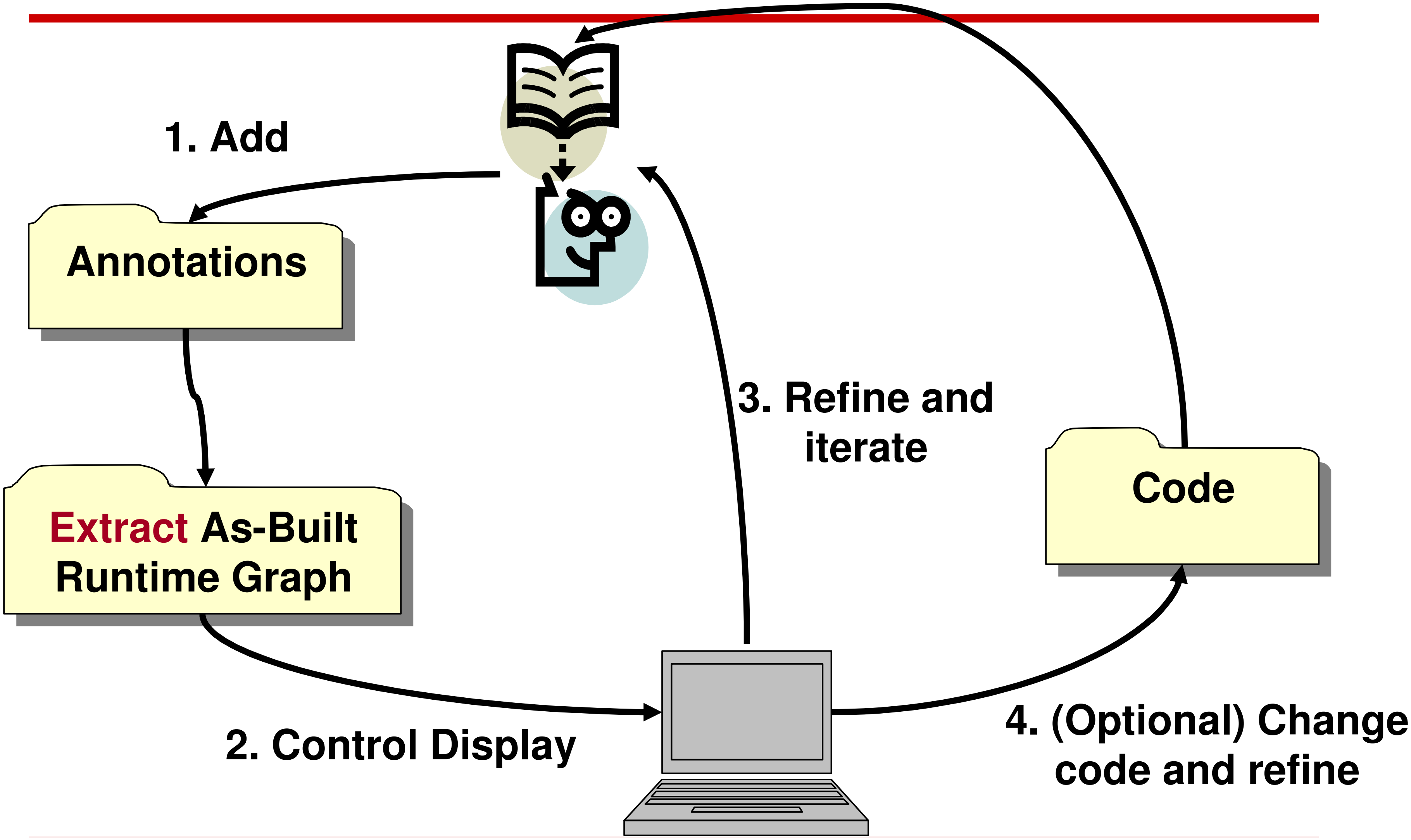


Tool Features

- Control projection depth
- Collapse/expand substructure
  - Selected domain or
  - Selected object
- Summary edges
- Elide private domains
- Control object labeling

Case Study: JHotDraw

Annotation/Extraction Process



1. Add

Annotations

Extract As-Built Runtime Graph

2. Control Display

3. Refine and iterate

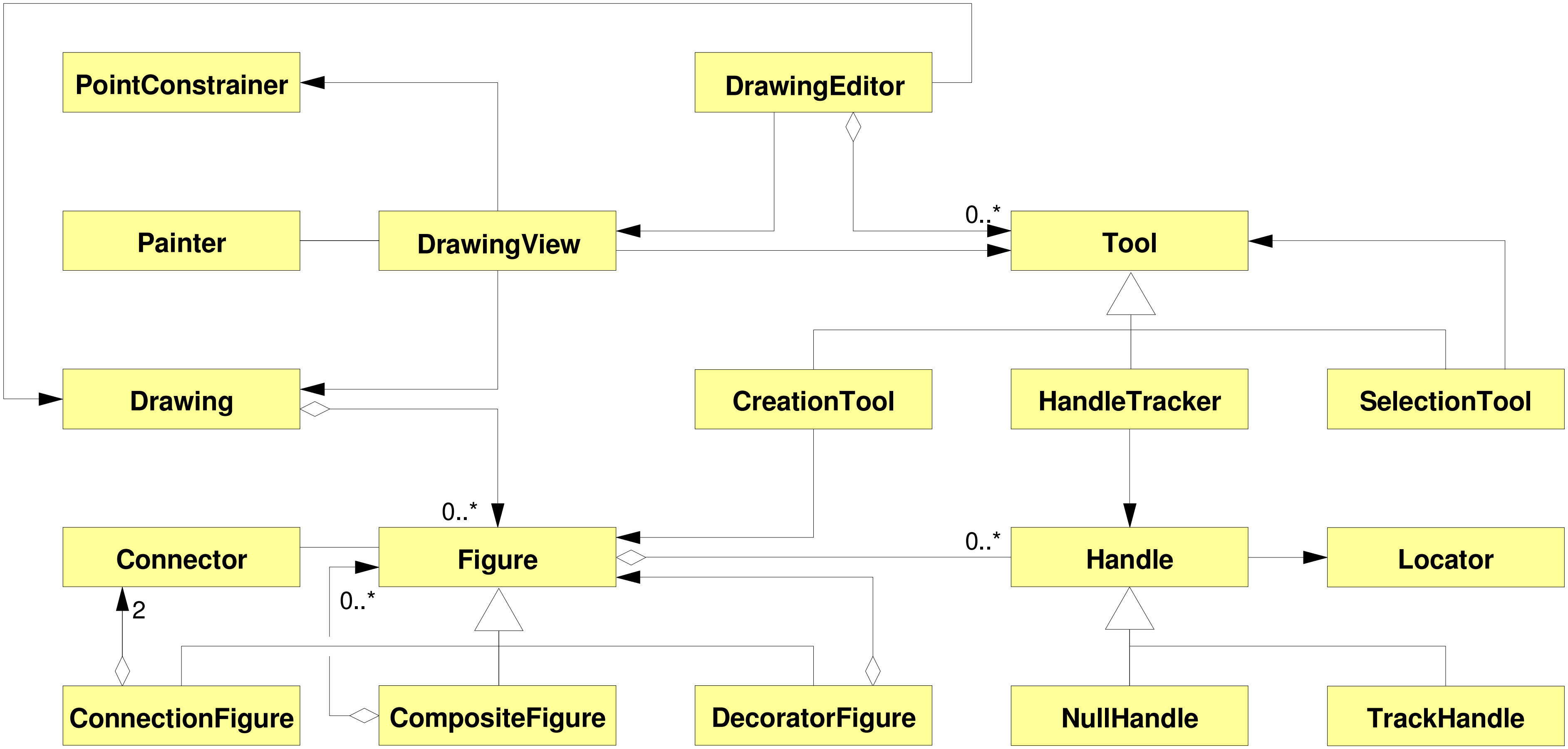
Code

4. (Optional) Change code and refine

Annotation/Extraction Process

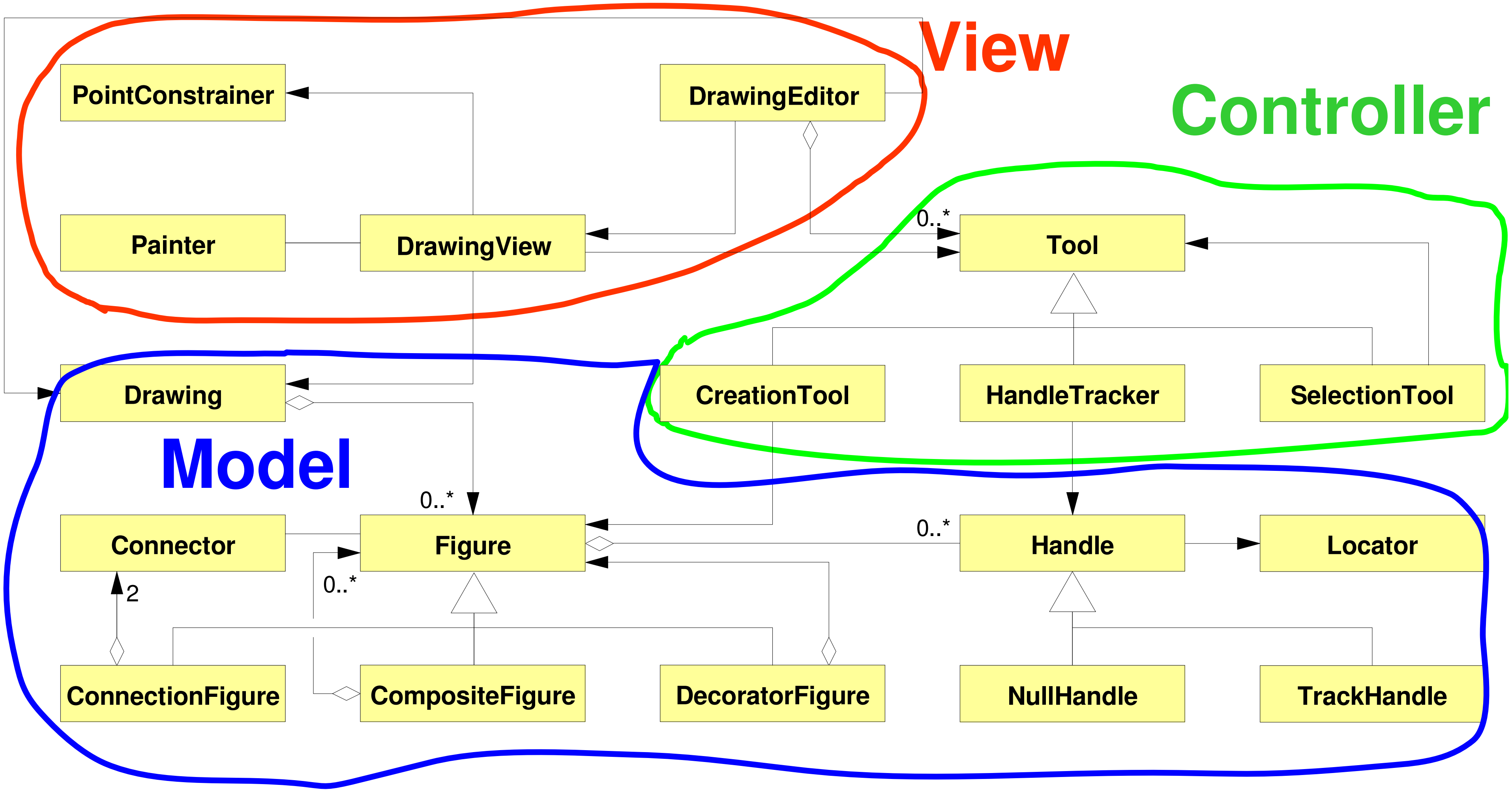
1. Choose **top-level domains**
2. Achieve **desired number of objects** in top-level domains
   a) Push **secondary objects under** primary objects
   b) Use **abstraction by types** to merge objects
3. Achieve appropriate **visual detail**
   a) Collapse or **expand substructure** of objects
   b) Change **projection depth** across all objects

JHotDraw: Code Structure

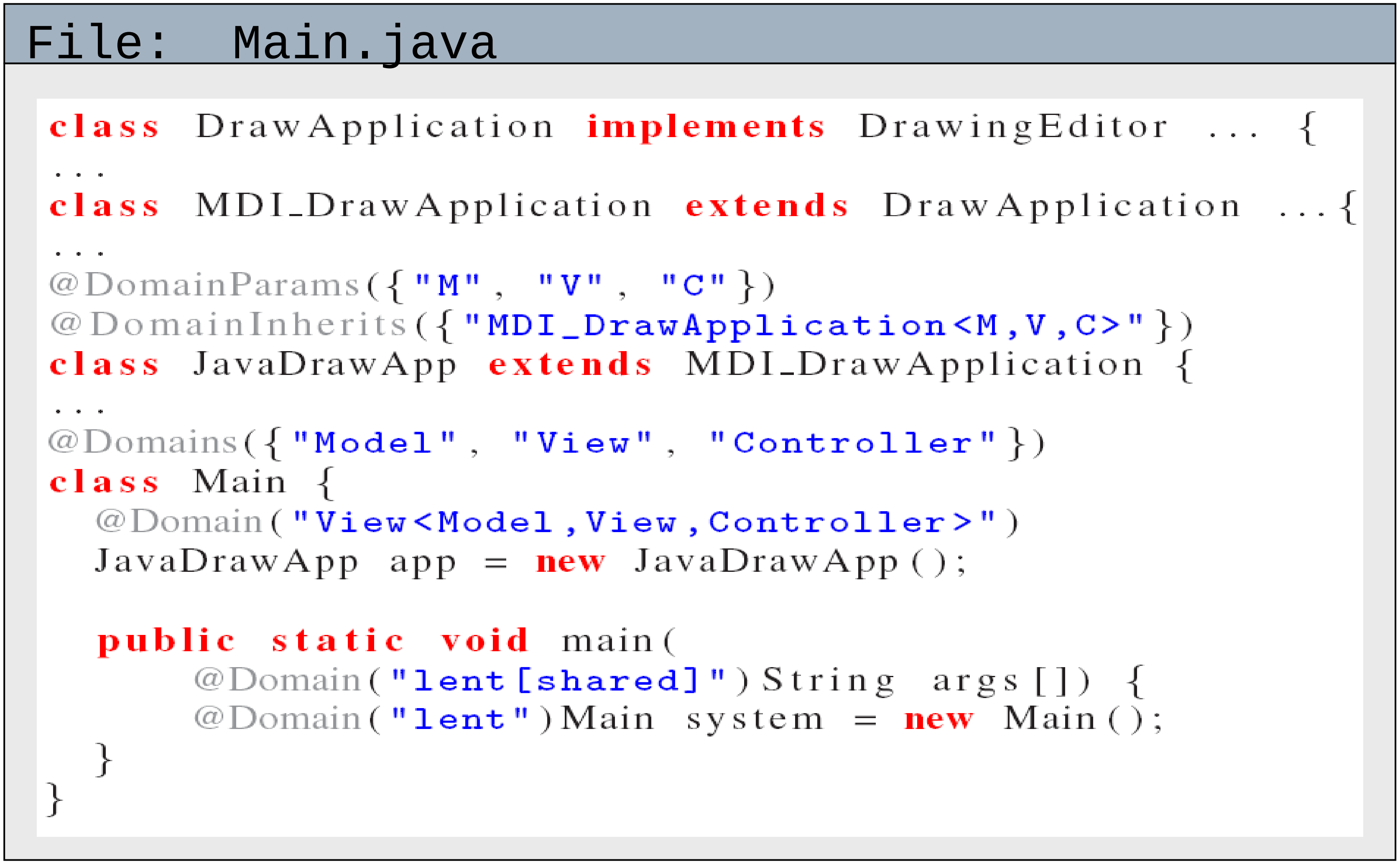


Manually generated UML Class Diagram for JHotDraw [Riehle, Thesis 2000].

JHotDraw: Model-View-Controller (MVC)



JHotDraw: Adding Annotations to Code

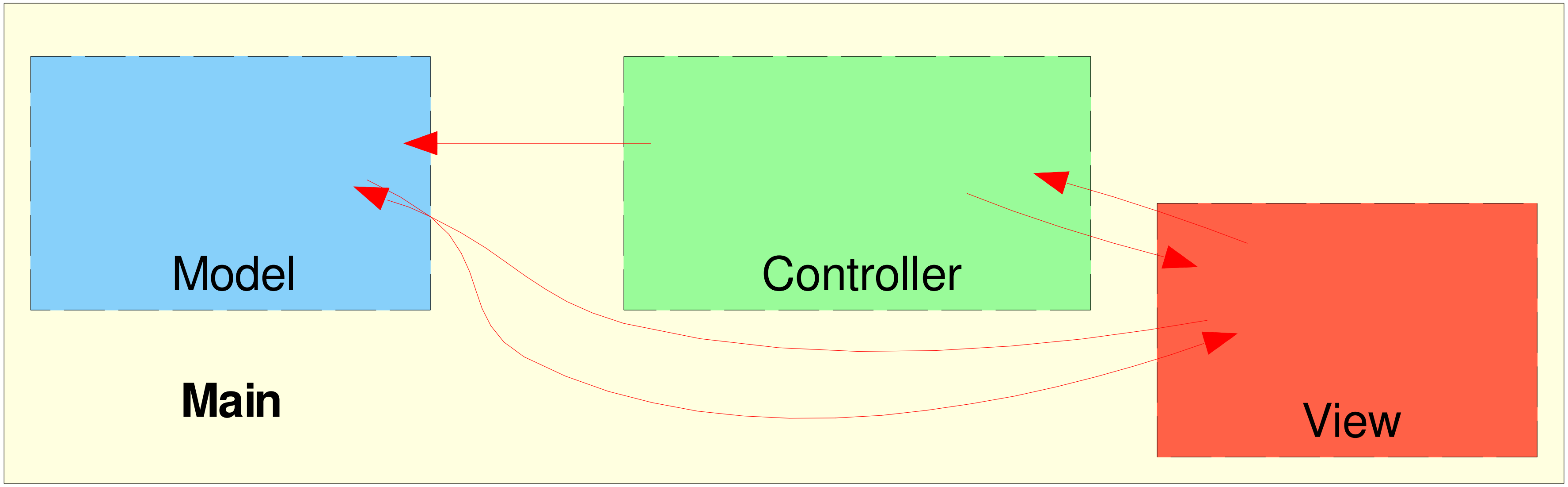


File: Main.java

```
class DrawApplication implements DrawingEditor ... {
...
class MDI_DrawApplication extends DrawApplication ...{
...
@DomainParams({"M", "V", "C"})
@DomainInherits({"MDI_DrawApplication<M,V,C>"})
class JavaDrawApp extends MDI_DrawApplication {
...
@Domains({"Model", "View", "Controller"})
class Main {
  @Domain("View<Model,View,Controller>")
  JavaDrawApp app = new JavaDrawApp();

  public static void main(
      @Domain("lent[shared]")String args[]) {
      @Domain("lent")Main system = new Main();
  }
}
```

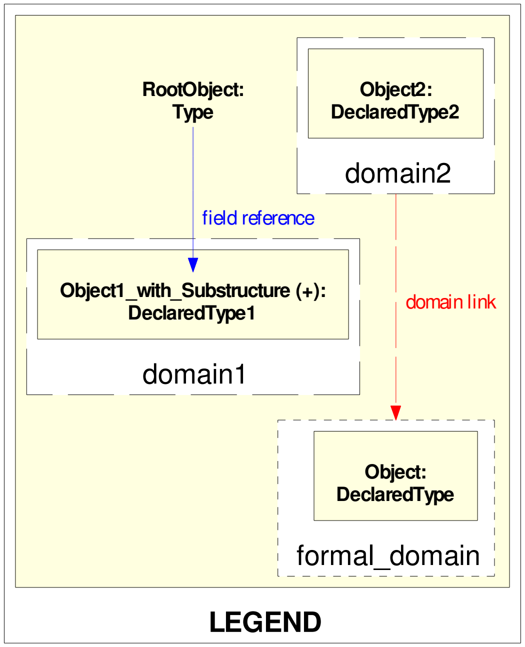
JHotDraw: “30-second Architecture”

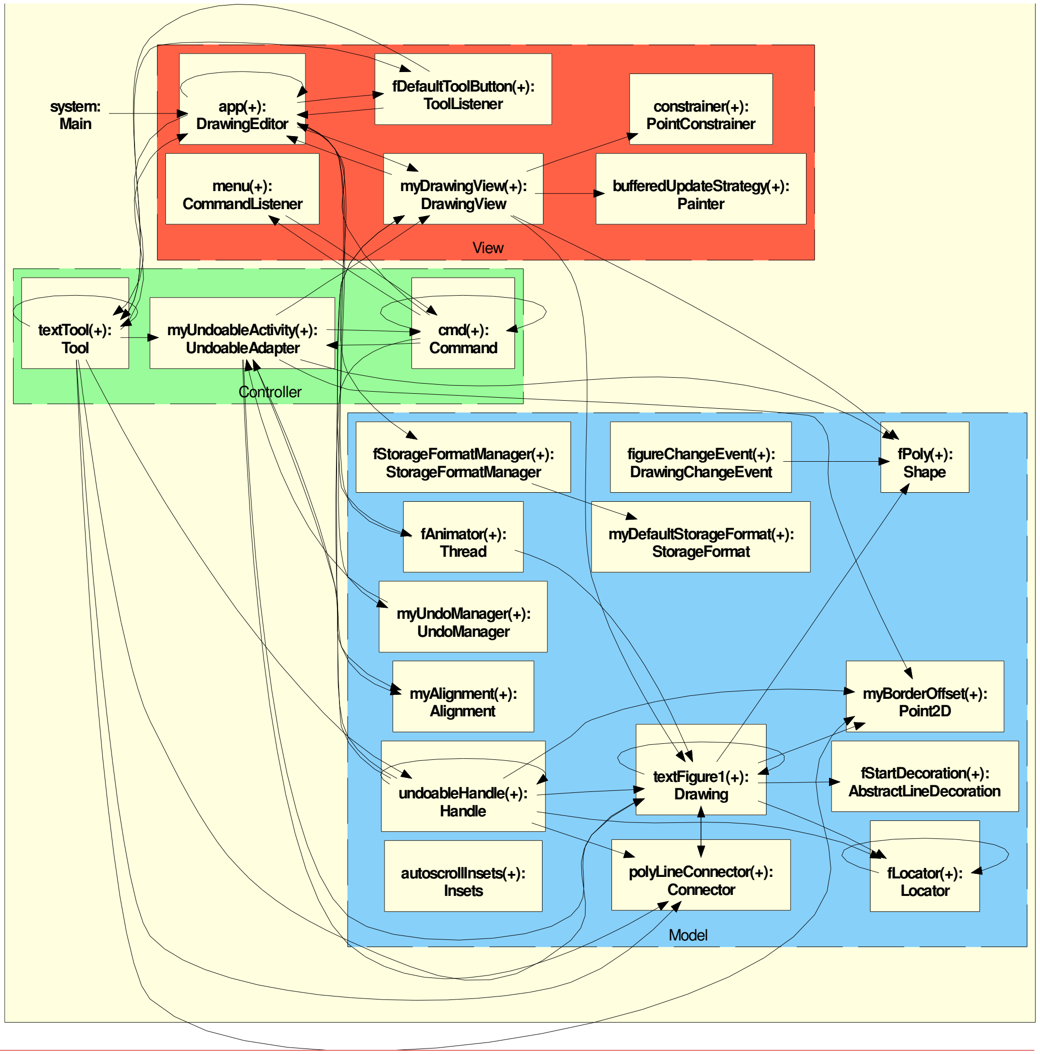
- Hide contents of domains
  - Dotted edges **summarize field references**
  - Interestingly: no callback from M to C



JHotDraw: “30-minute Architecture”

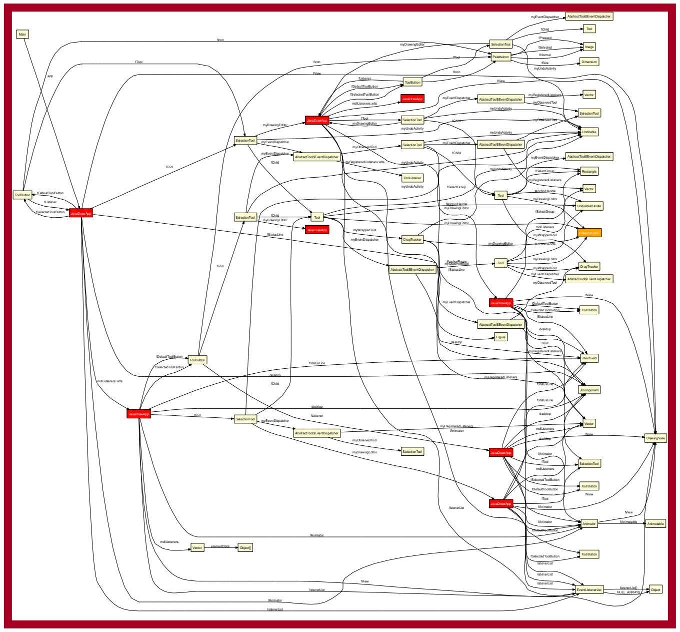
Showing top-level domains and objects



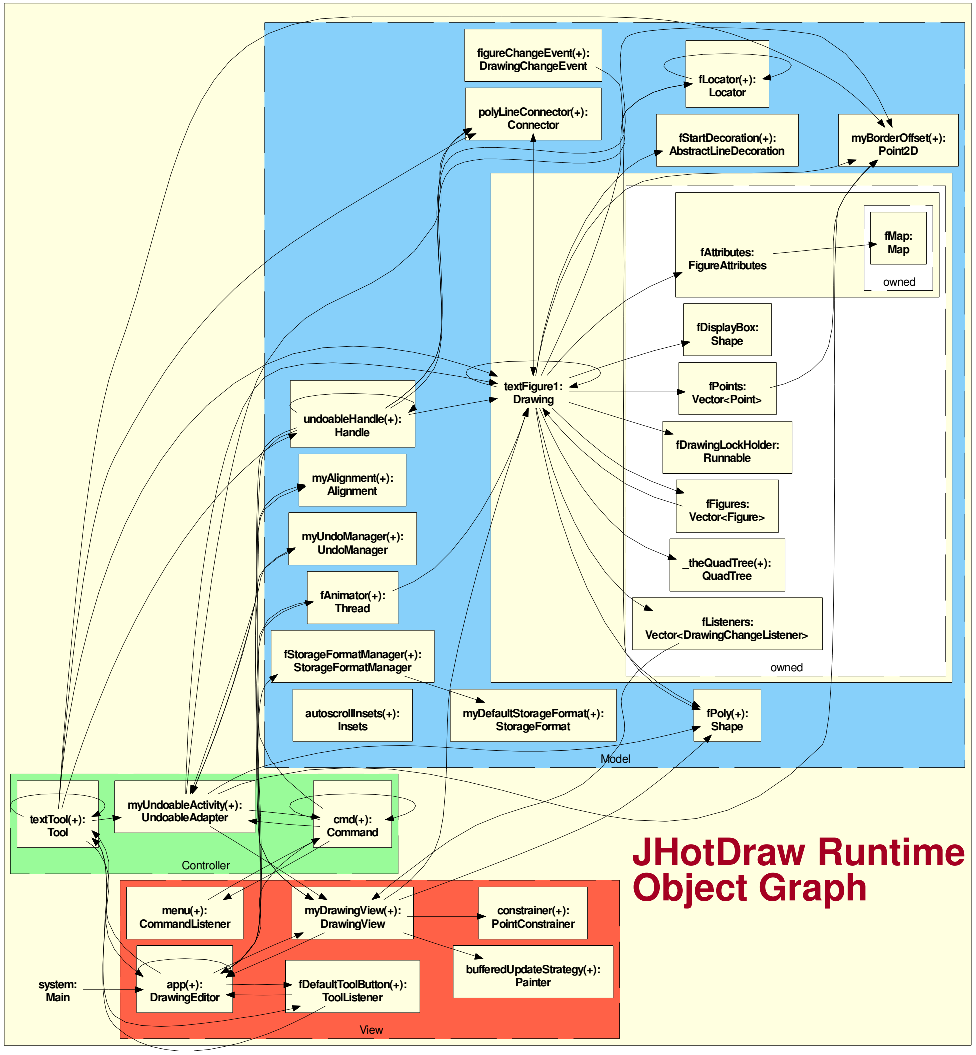


JHotDraw: “30-minute Architecture”

Showing `Drawing`’s sub-structure



Output of Womble

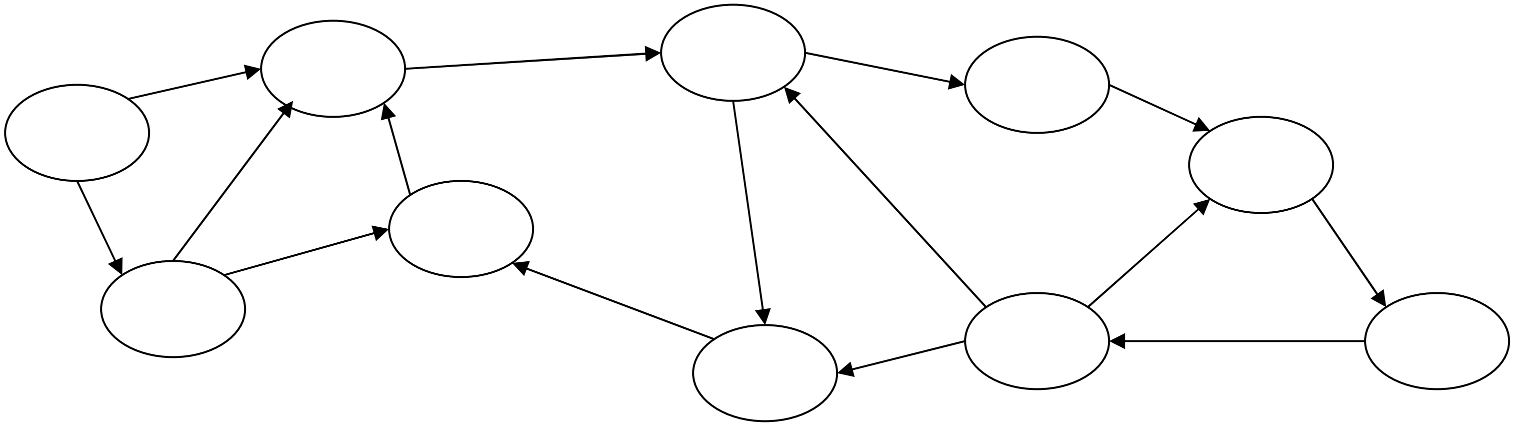


JHotDraw Runtime Object Graph

Static Analysis

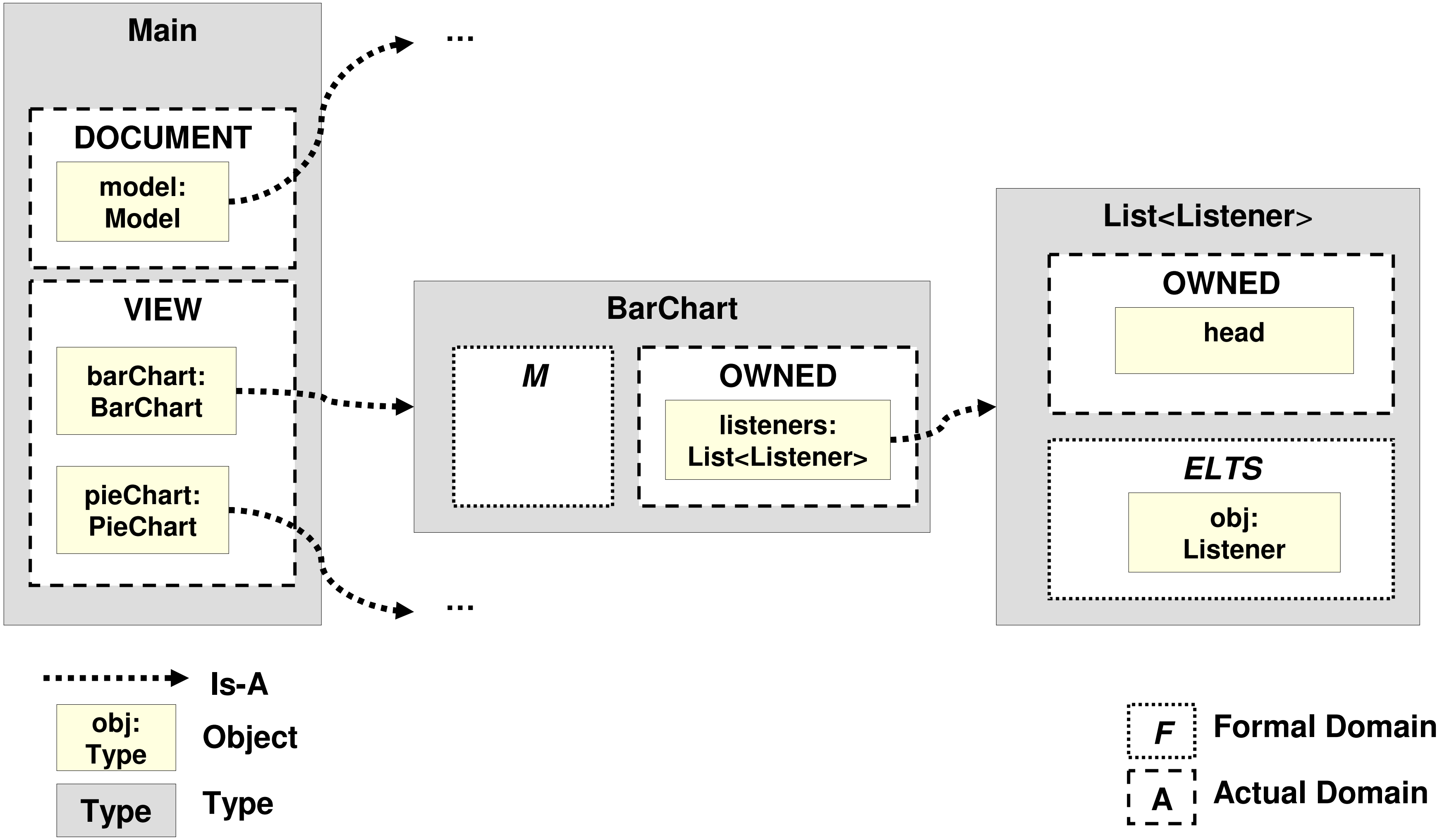
Static analysis

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

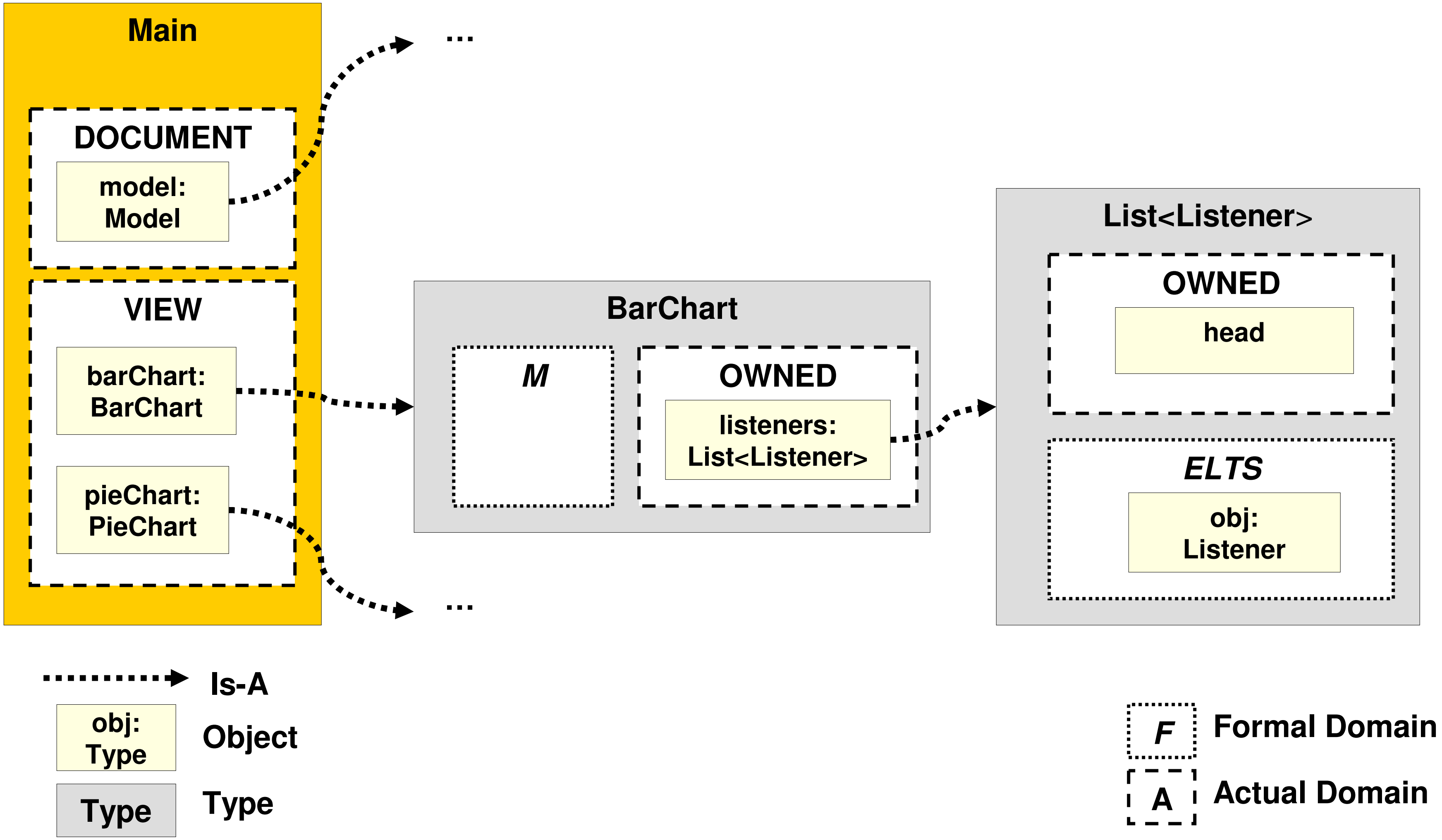


ROG: graph where nodes represent runtime objects, edges represent reference or usage relations

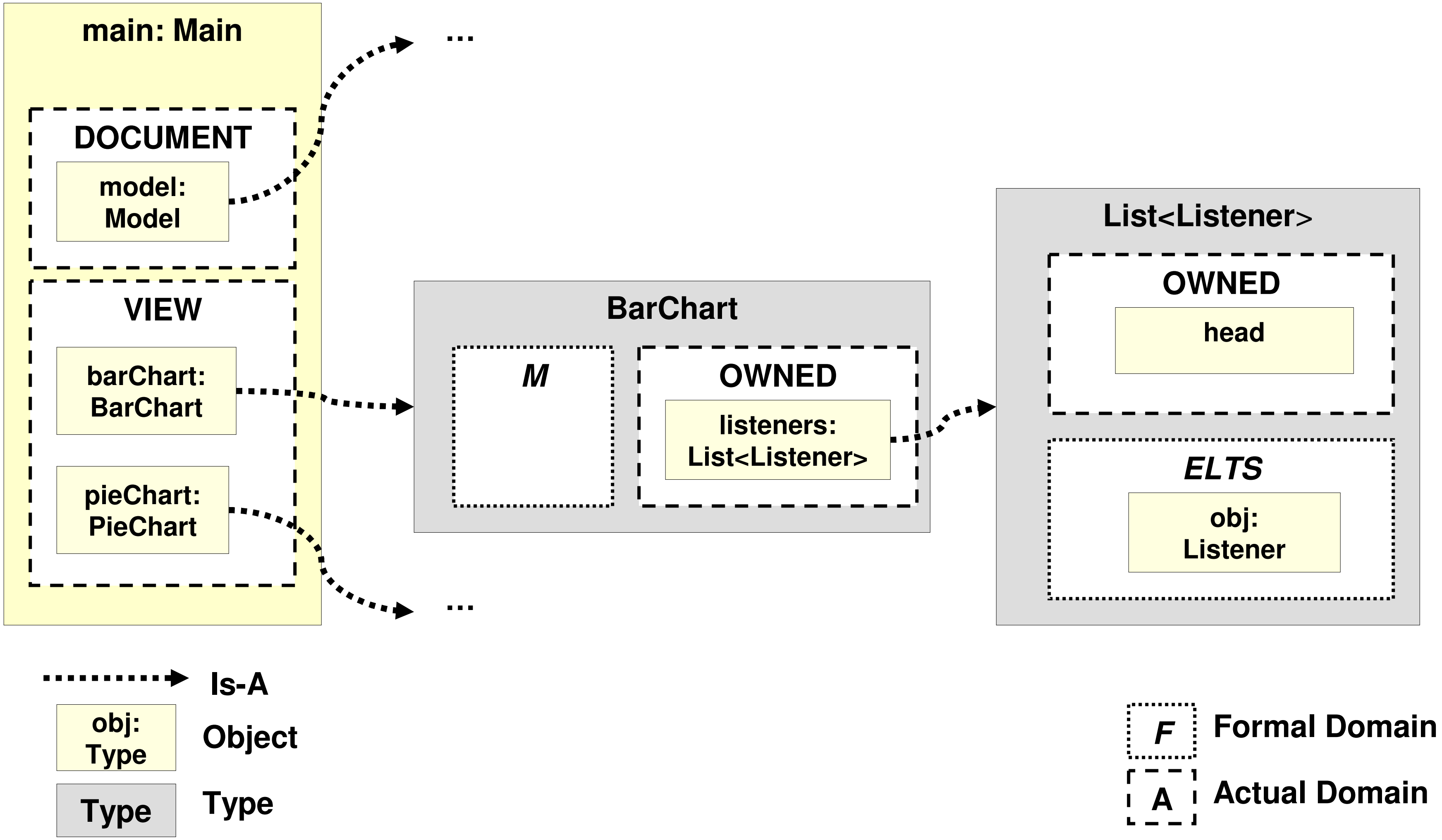
TypeGraph: show types, domains inside types, and objects in domains



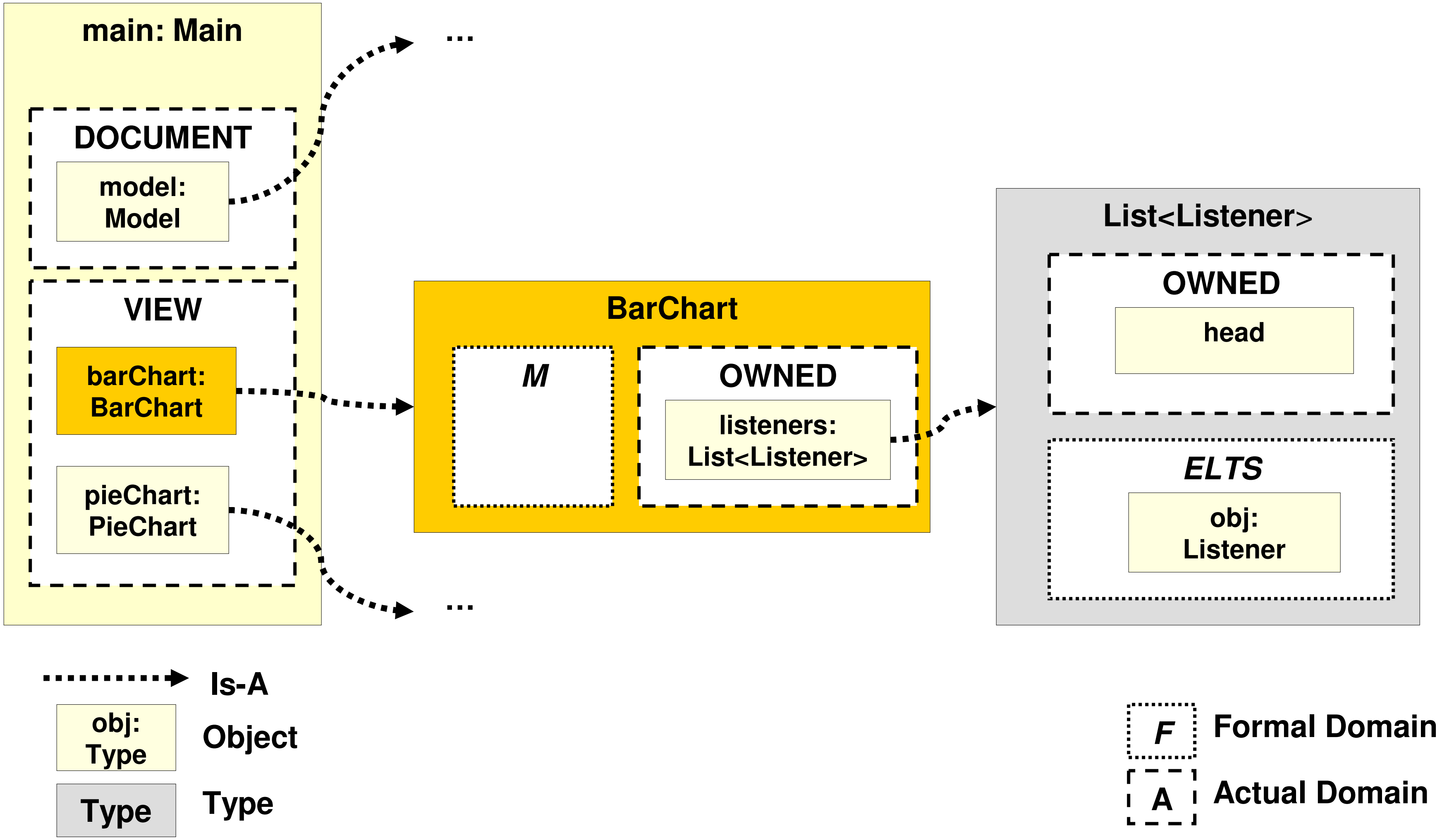
ObjectGraph: instantiate types, starting with root



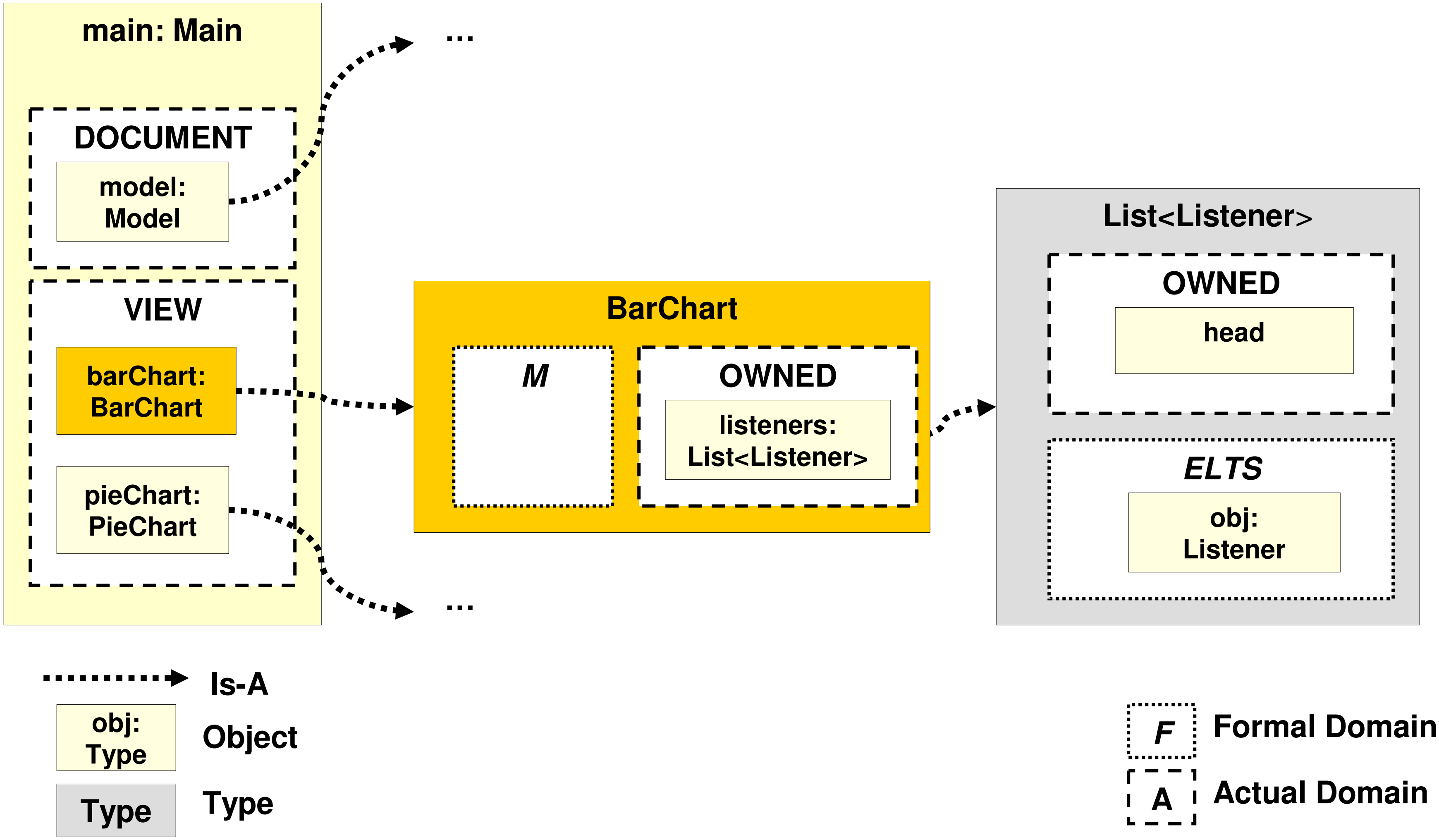
ObjectGraph: instantiate types, starting with root



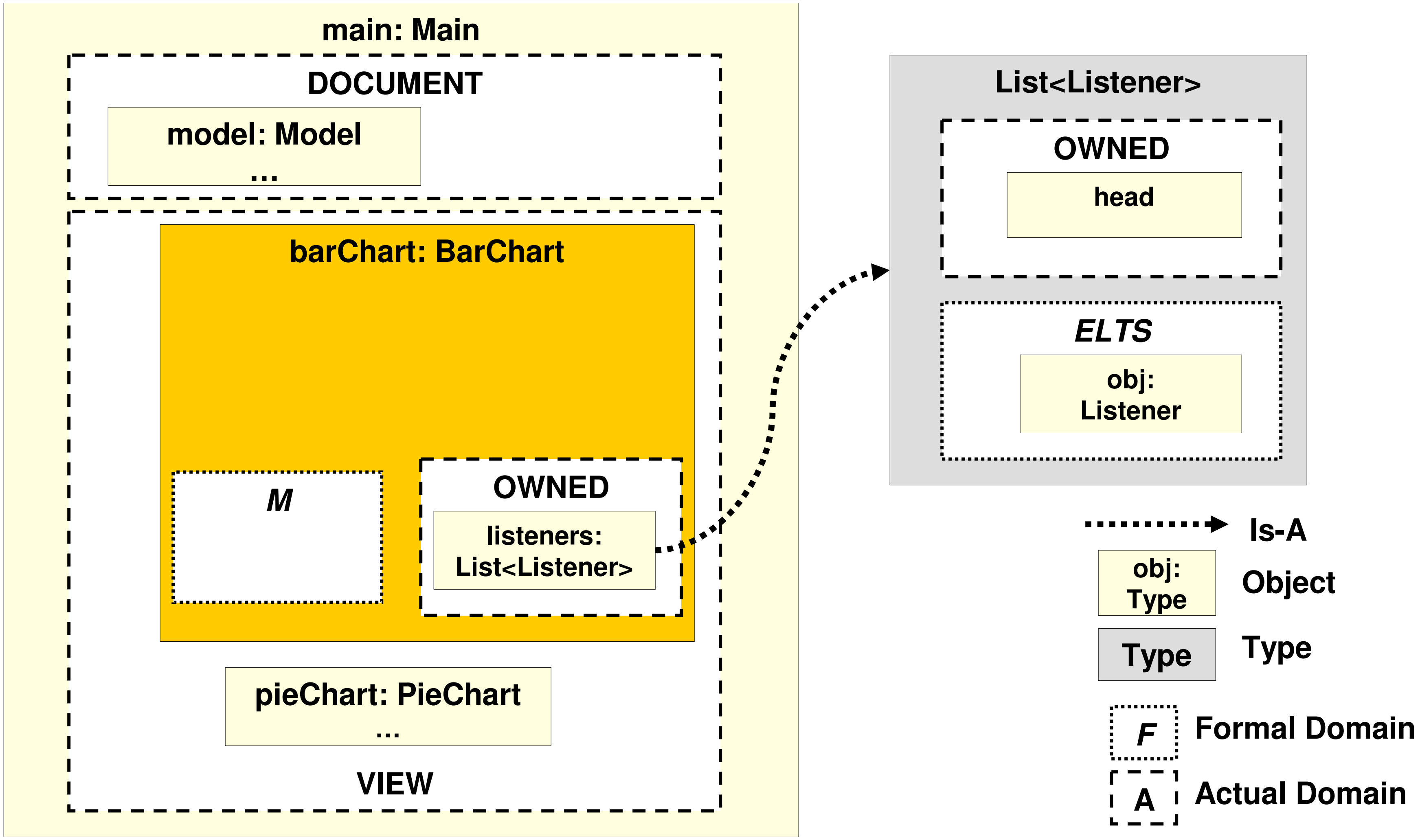
ObjectGraph: instantiate types, show domains and objects inside domains



ObjectGraph: instantiate types, show domains and objects inside domains



ObjectGraph: instantiate types, show domains and objects inside domains



main: Main

DOCUMENT

model: Model
…

barChart: BarChart

M

OWNED

listeners:
List<Listener>

pieChart: PieChart
…

VIEW

List<Listener>

OWNED

head

ELTS

obj:
Listener

Is-A

obj:
Type

Object

Type

Type

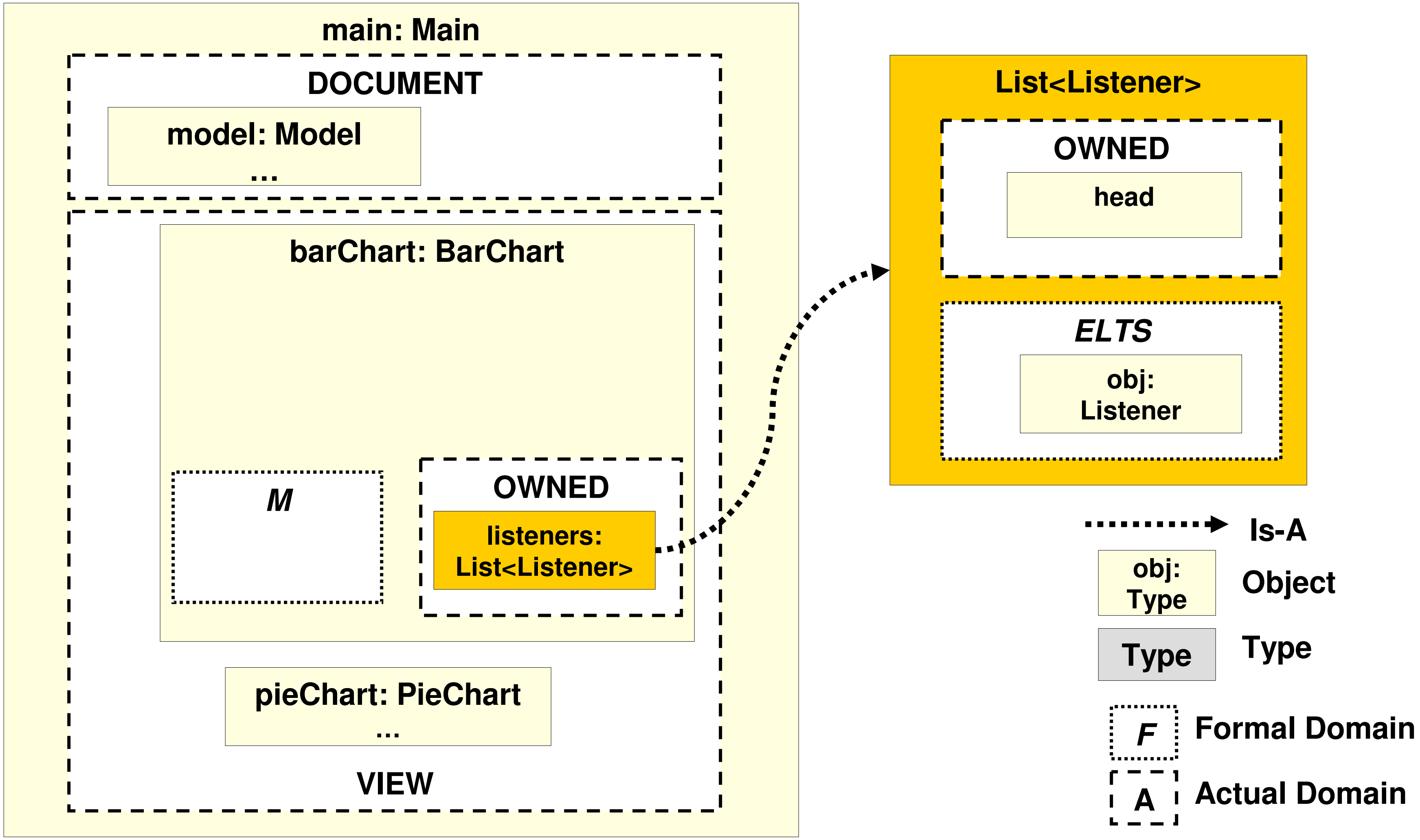
F

Formal Domain

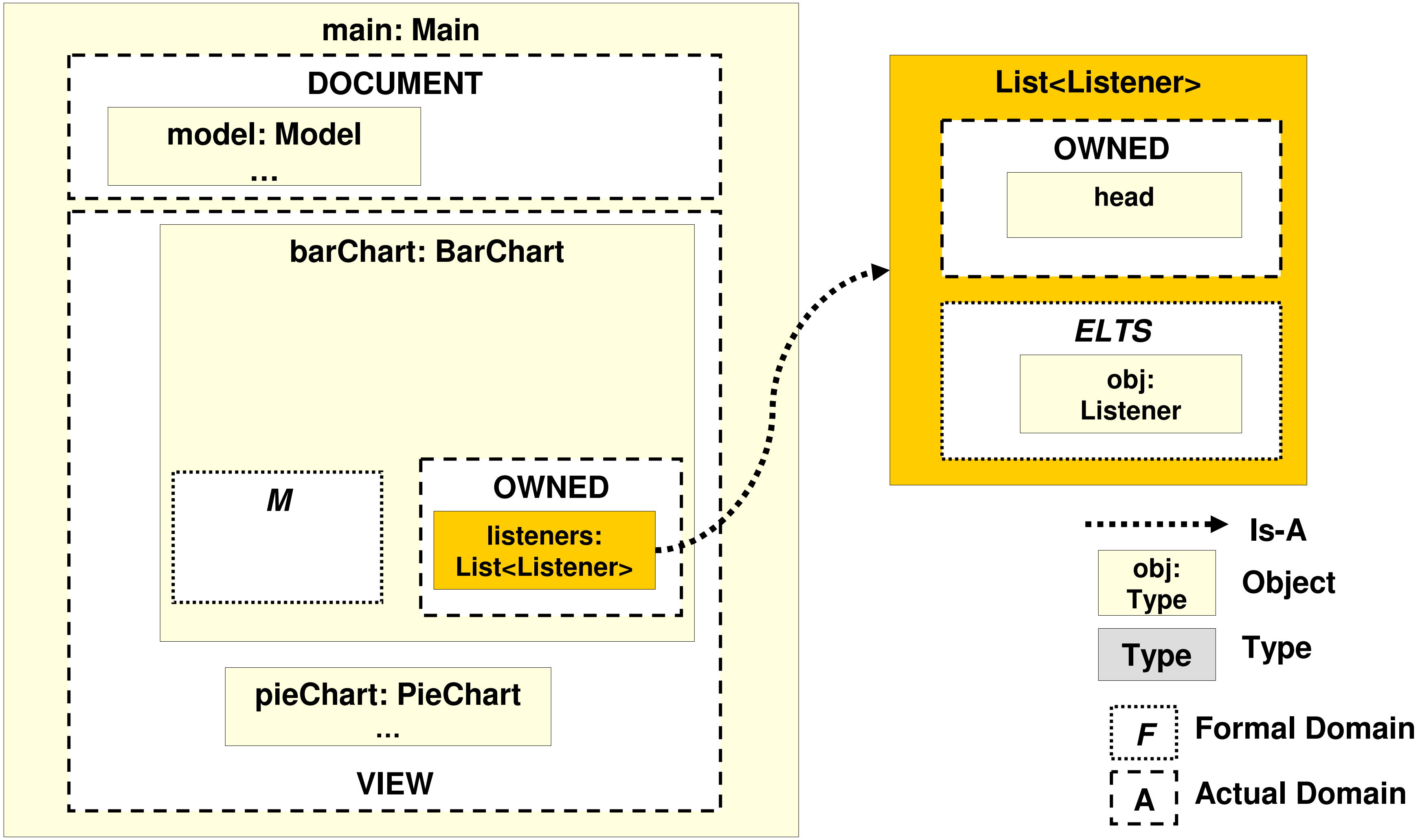
A

Actual Domain

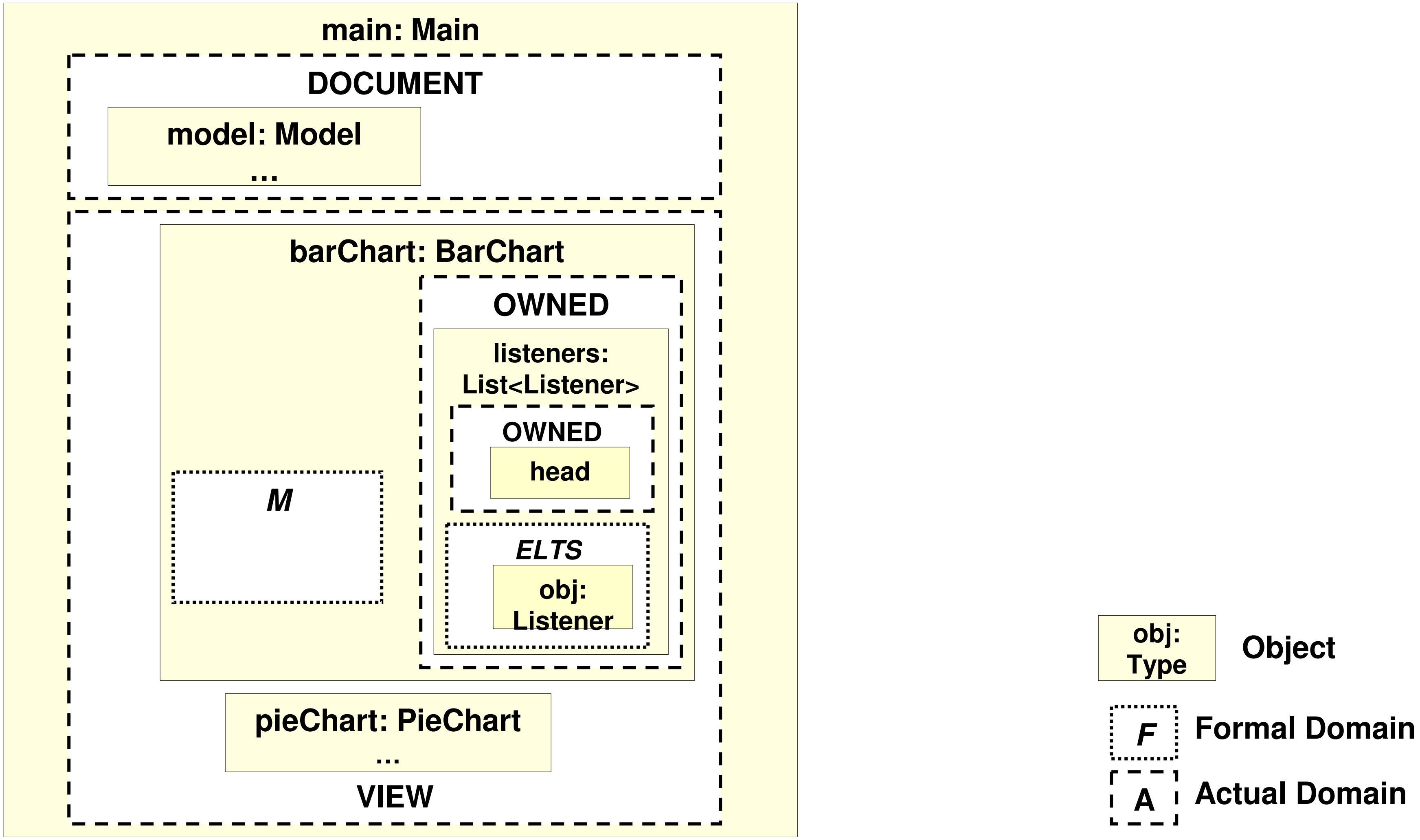
ObjectGraph: instantiate types, show domains and objects inside domains



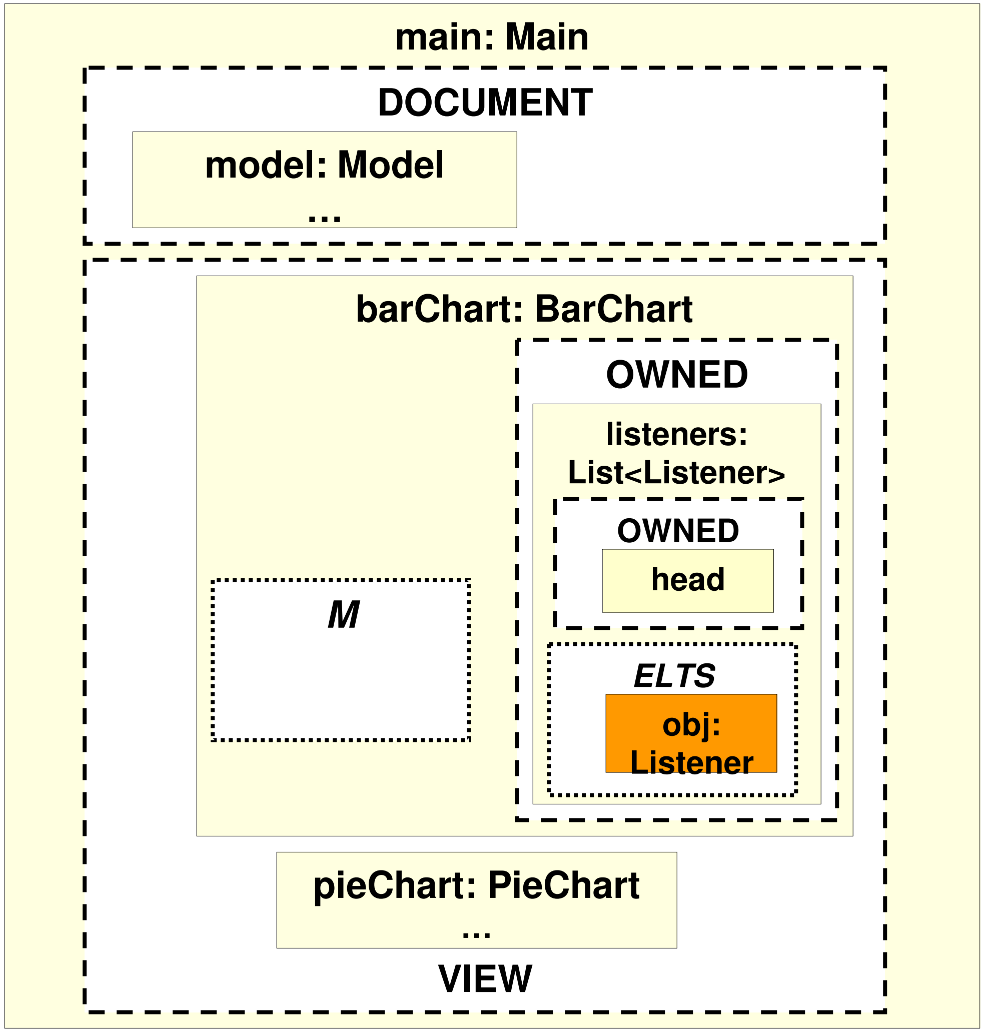
ObjectGraph: instantiate types, show domains and objects inside domains

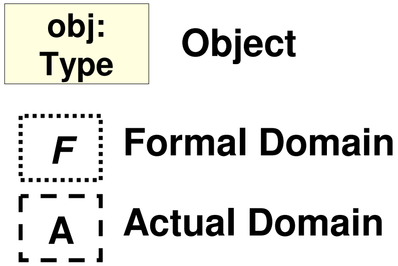


ObjectGraph: instantiate types, show domains and objects inside domains

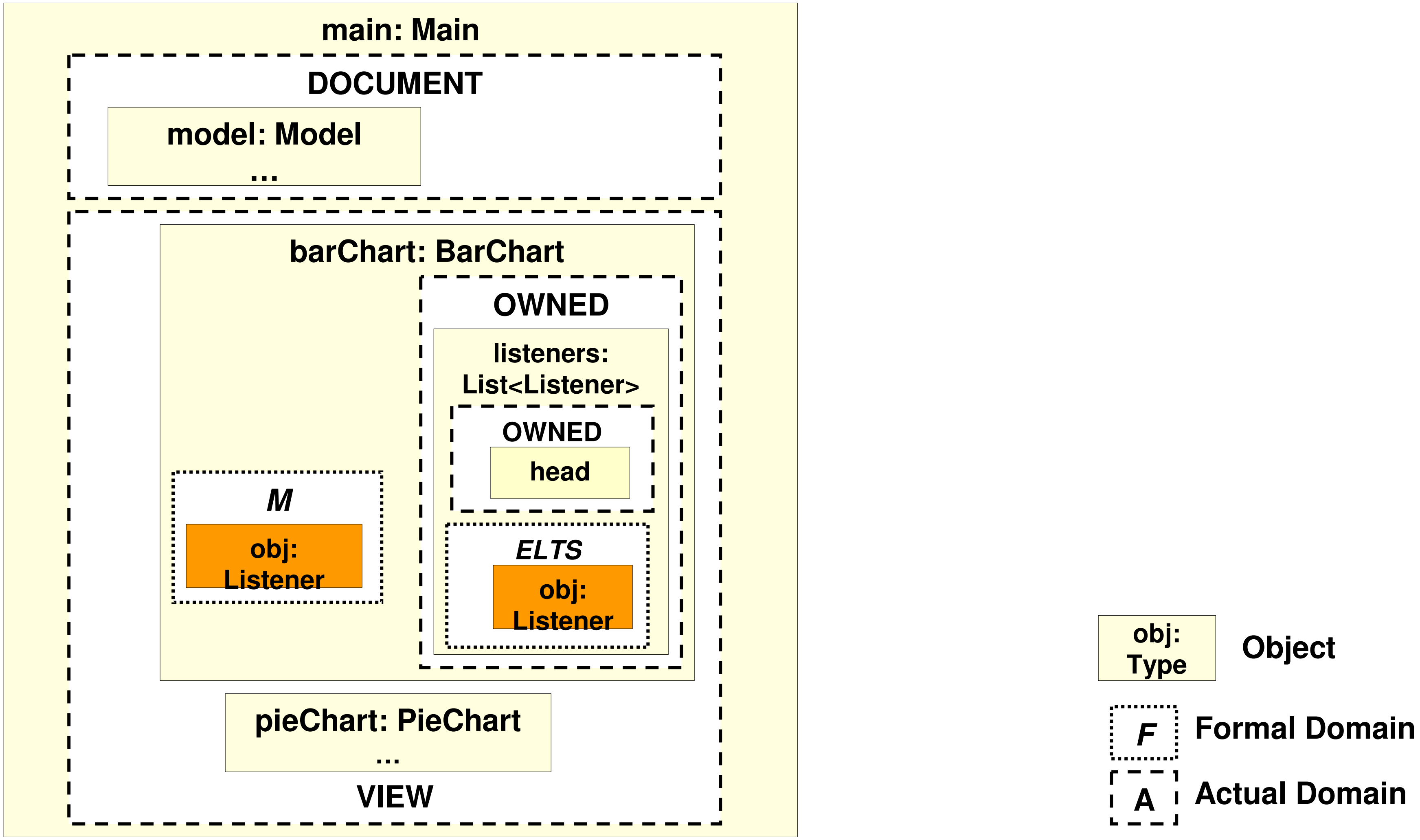


ObjectGraph: pull objects from formal domains to actual domains

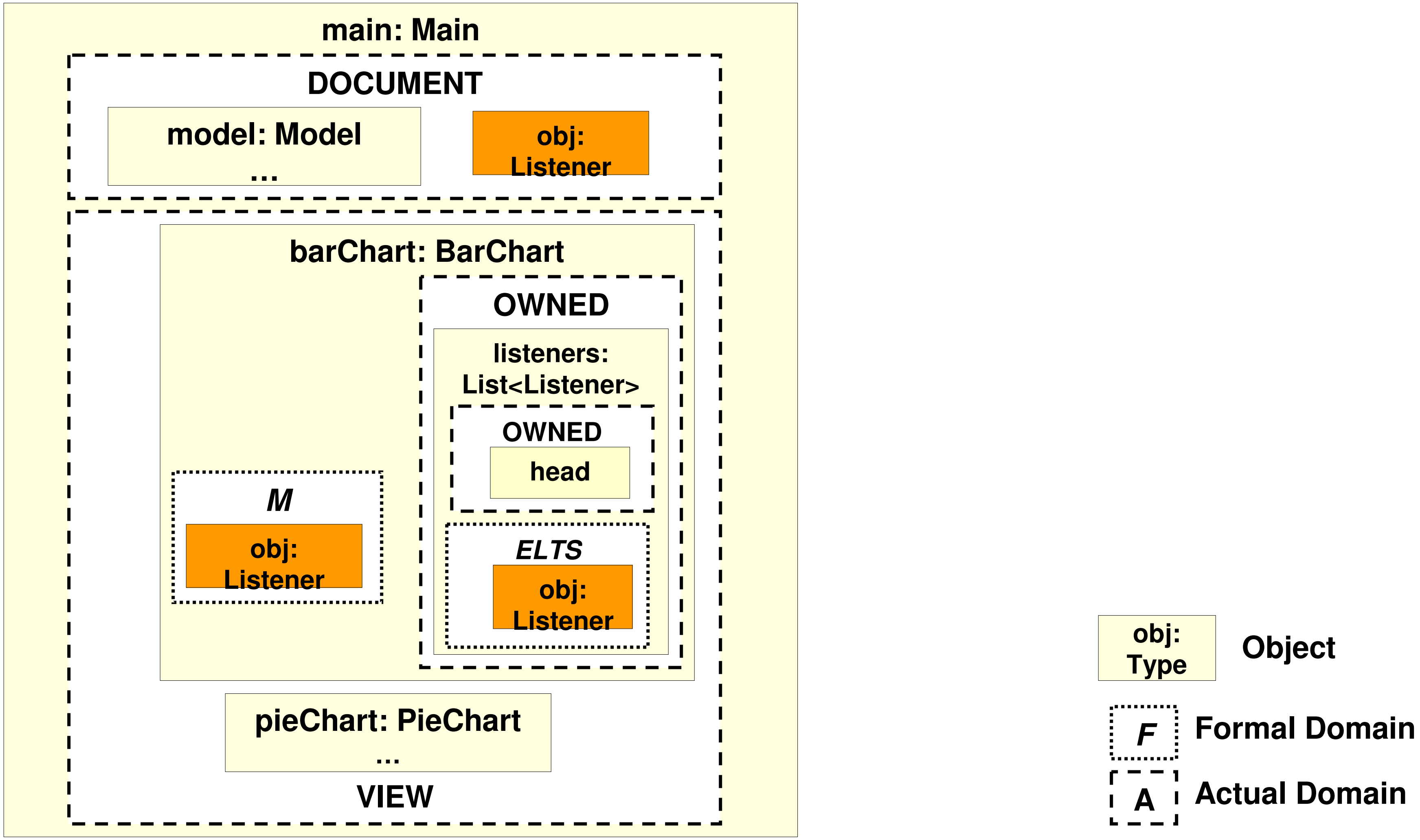




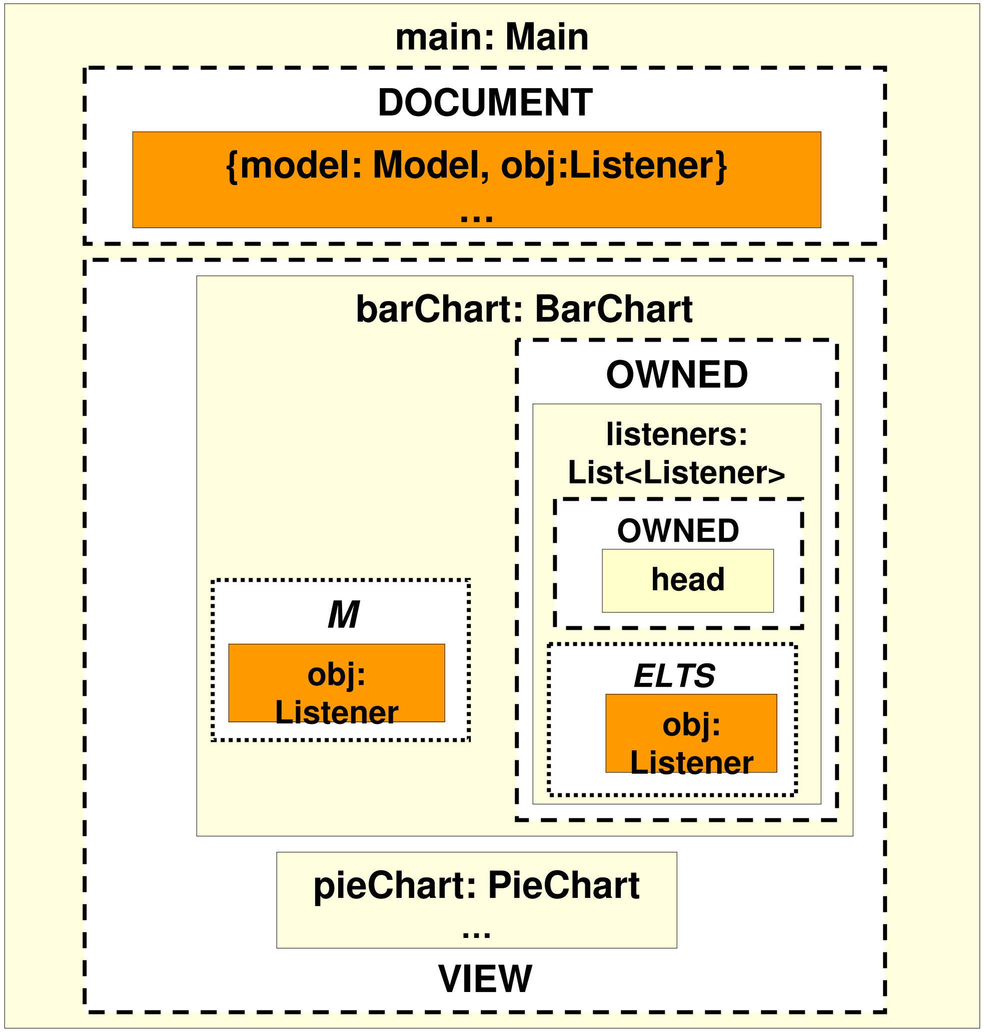
ObjectGraph: pull objects from formal domains to actual domains



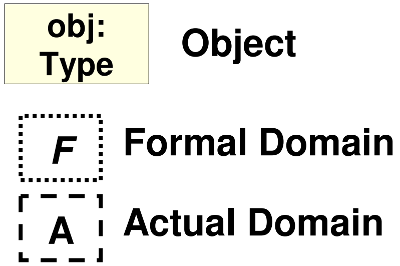
ObjectGraph: pull objects from formal domains to actual domains



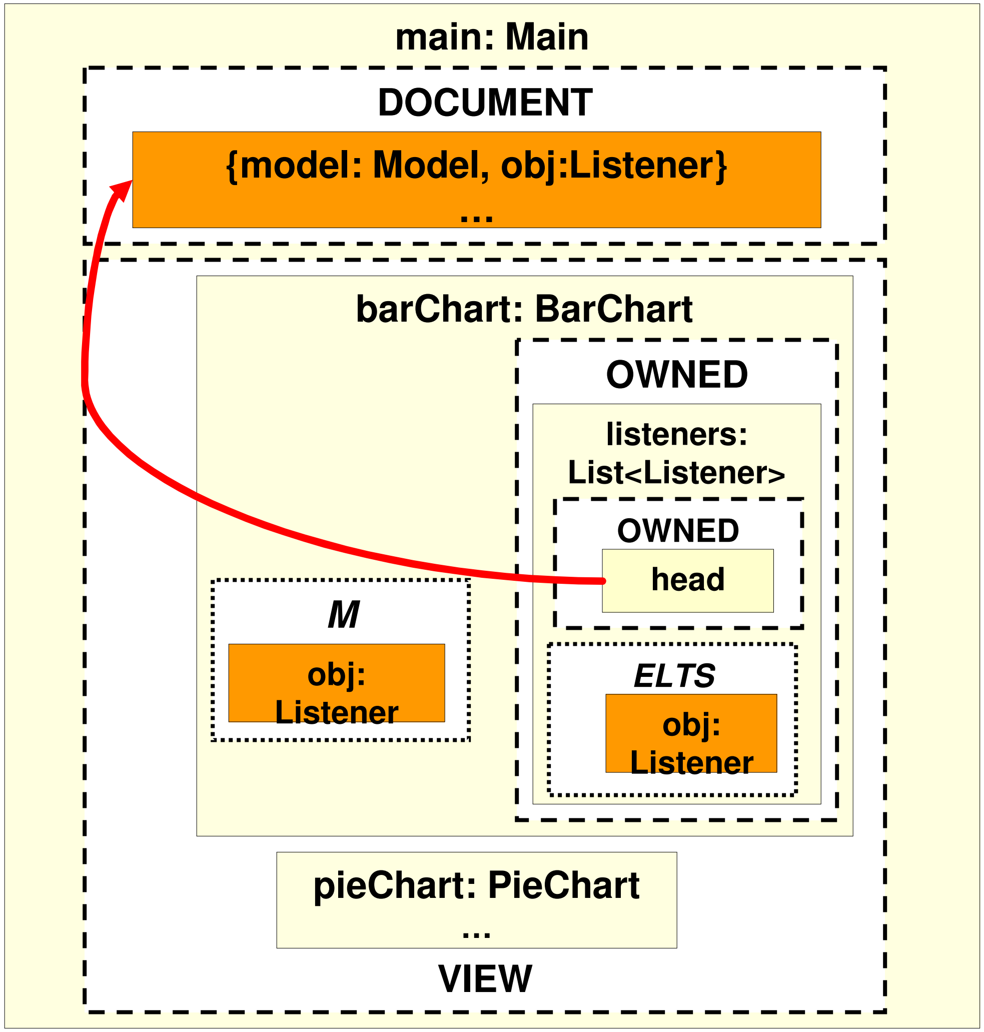
ObjectGraph: merge objects, in one domain, that *may* alias, based on types

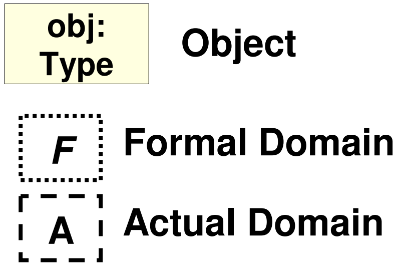


```
class Model implements Listener {
…
}
```



ObjectGraph: add edges to represent field references





Conclusion

- **Ownership domain annotations** enable extraction of **hierarchical** runtime object graphs using **static analysis**
- Provide architectural abstraction by ownership hierarchy and by types