

*International Workshop on Aliasing, Confinement and
Ownership in object-oriented programming (IWACO)*

Compile-Time Views of Execution Structure based on Ownership

*“Inside every large program is a small program struggling to get out”
– C.A.R. Hoare, Efficient Production of Large Programs (1970)*

Marwan Abi-Antoun

Jonathan Aldrich

School of Computer Science
Carnegie Mellon University, USA



Fact: object-oriented code is hard to understand

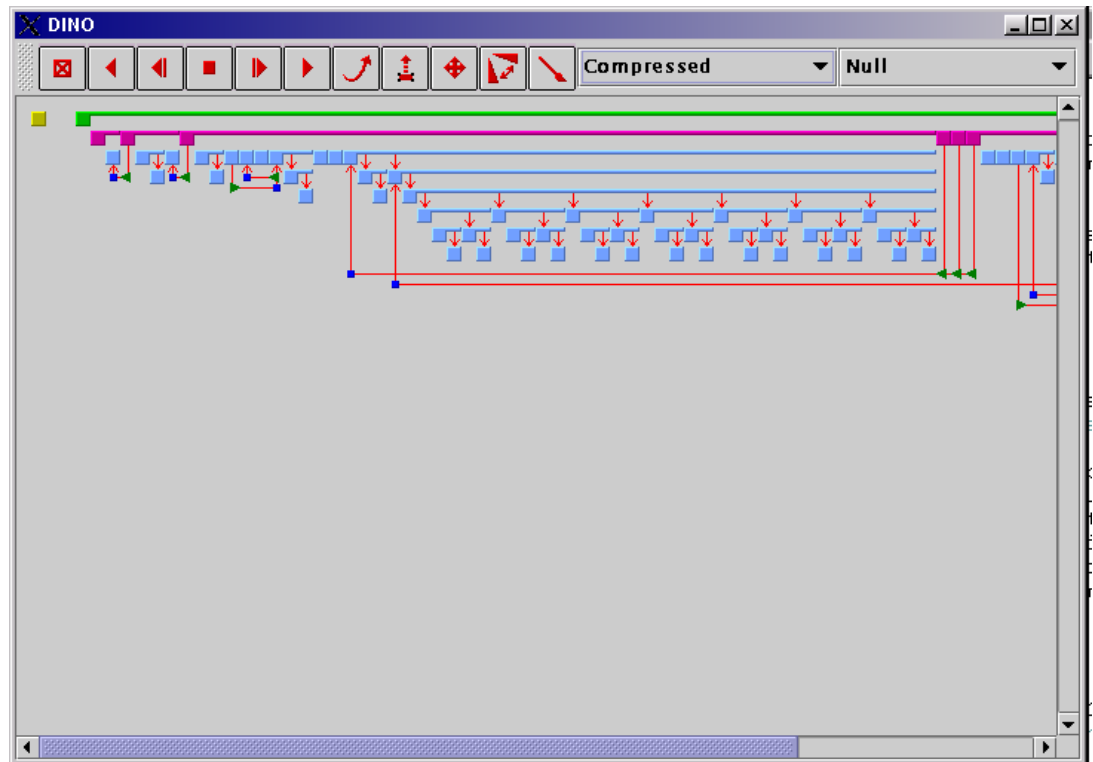
- Dichotomy between two design structures
- **Code structure**
 - Hierarchies of classes frozen at ***compile-time***
 - E.g., UML class diagrams
 - (a.k.a. *static* structure)
 - Many tools available
- **Execution structure**
 - Dynamic networks of objects at ***run-time***
 - E.g., UML object diagrams
 - (a.k.a. *runtime* structure, *dynamic* structure)
 - Tool support much less mature

Dynamic analyses to obtain execution structure

- Produce traces that get filtered, queried and visualized in novel ways
- Limitations of dynamic analyses
 - Show execution structure for a program run
 - Do not convey design intent
 - Cannot deal with incomplete programs
 - High runtime overhead in some cases
- Hierarchical graphs more scalable than flat graphs

Using ownership for adding hierarchy to the runtime object graph

- Lightweight ownership inference
- Strict *owners-as-dominators*
 - Abstraction
 - Encapsulation



Output of DINO (Noble, Potter et al., 2002)

Challenges of obtaining the Execution Structure at compile-time

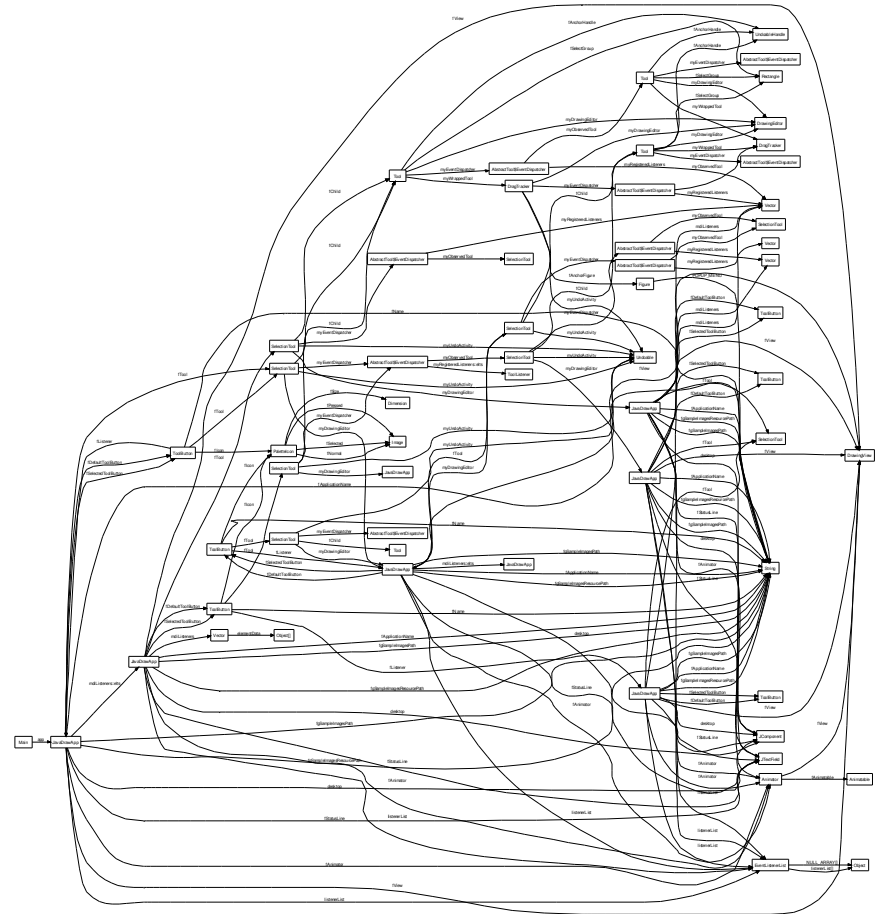
- Design intent
 - Group runtime objects into meaningful clusters
- Scalability (both analysis and visualization)
 - Group runtime objects into fewer top-level “abstract” elements
 - Output should be readable
- Hierarchy
 - Need to attain both high-level understanding and detail
- Aliasing
 - Can these two variables be aliased?
- Precision
 - What is the exact type of this variable?
 - How many instances of these may exist?
- Soundness
 - Reveal all relationships that may exist at runtime

Existing static analyses

- Can handle source code without annotations
- Work on bytecode in some cases,
- Heavy weight analyses
 - Precise but unscalable [OCa01]
- Produce flat (non-hierarchical) graphs
 - Do not scale [JW01, OCa01, Spi02]
- Fail to reveal relations that exist
 - Unsound analyses [JW01, Spi02]
- Annotations just to obtain visualization
 - [LR03]
- Do not provide design intent

Execution structure using static tool: JHotDraw as seen by Womble

- JHotDraw v. 5.3
 - 15,000 LOC
- Does not fit on one **readable** page
- **Unsound** analysis

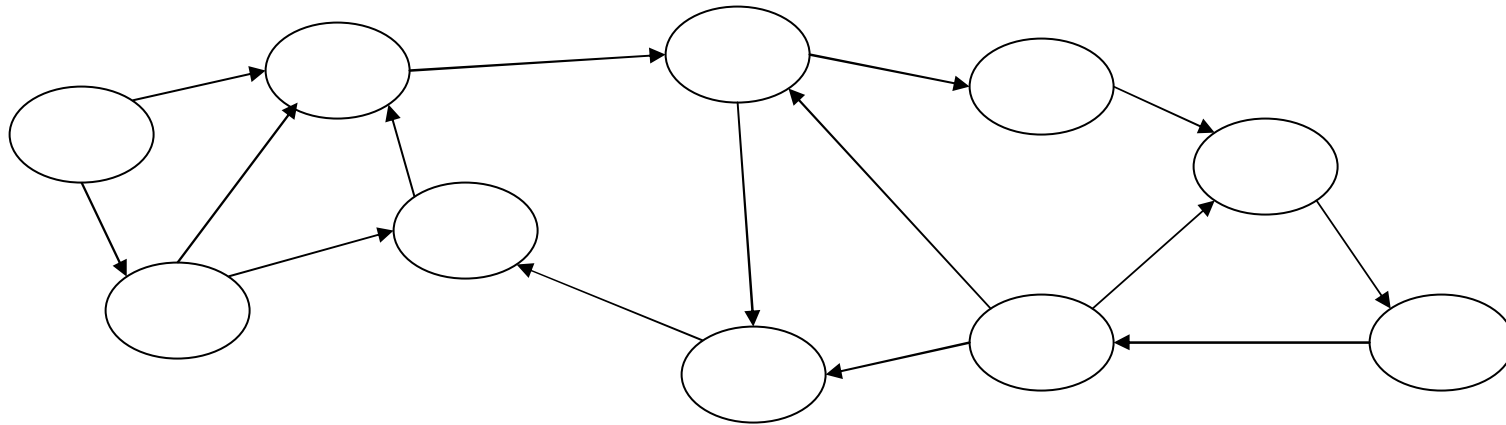


Output of Womble (Jackson and Waingold [JW01])

Execution Structure based on Ownership

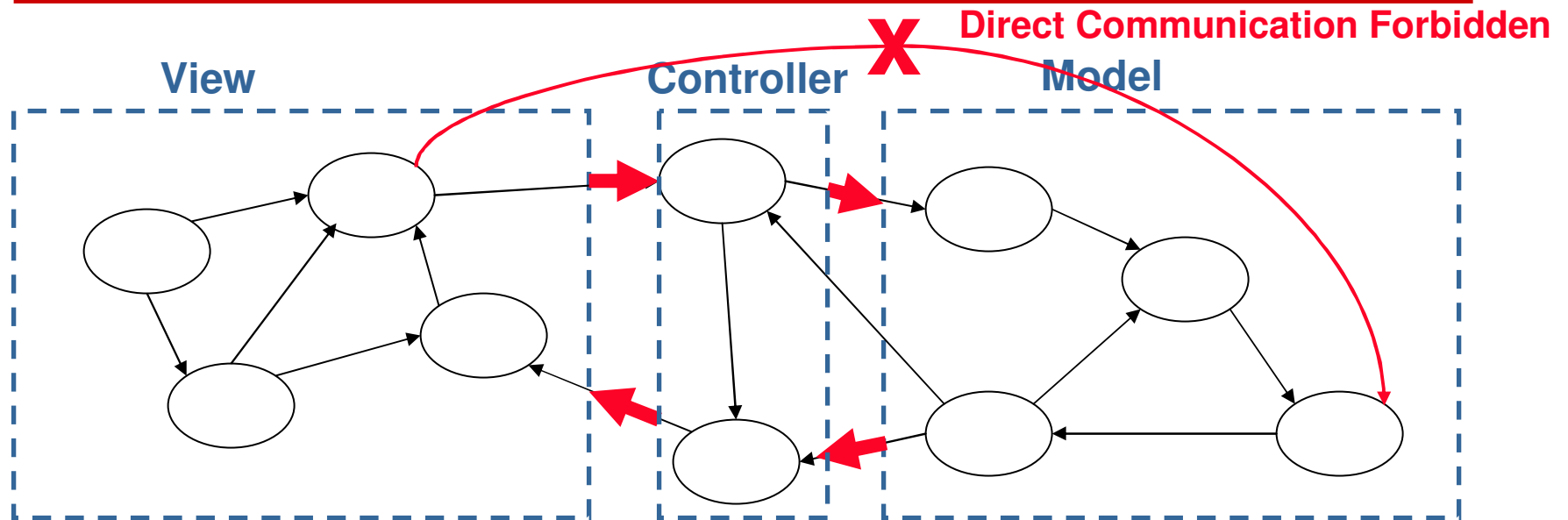
- Introduction
- **Approach**
- Details
- Validation

How to approximate the runtime Object Graph at compile-time?



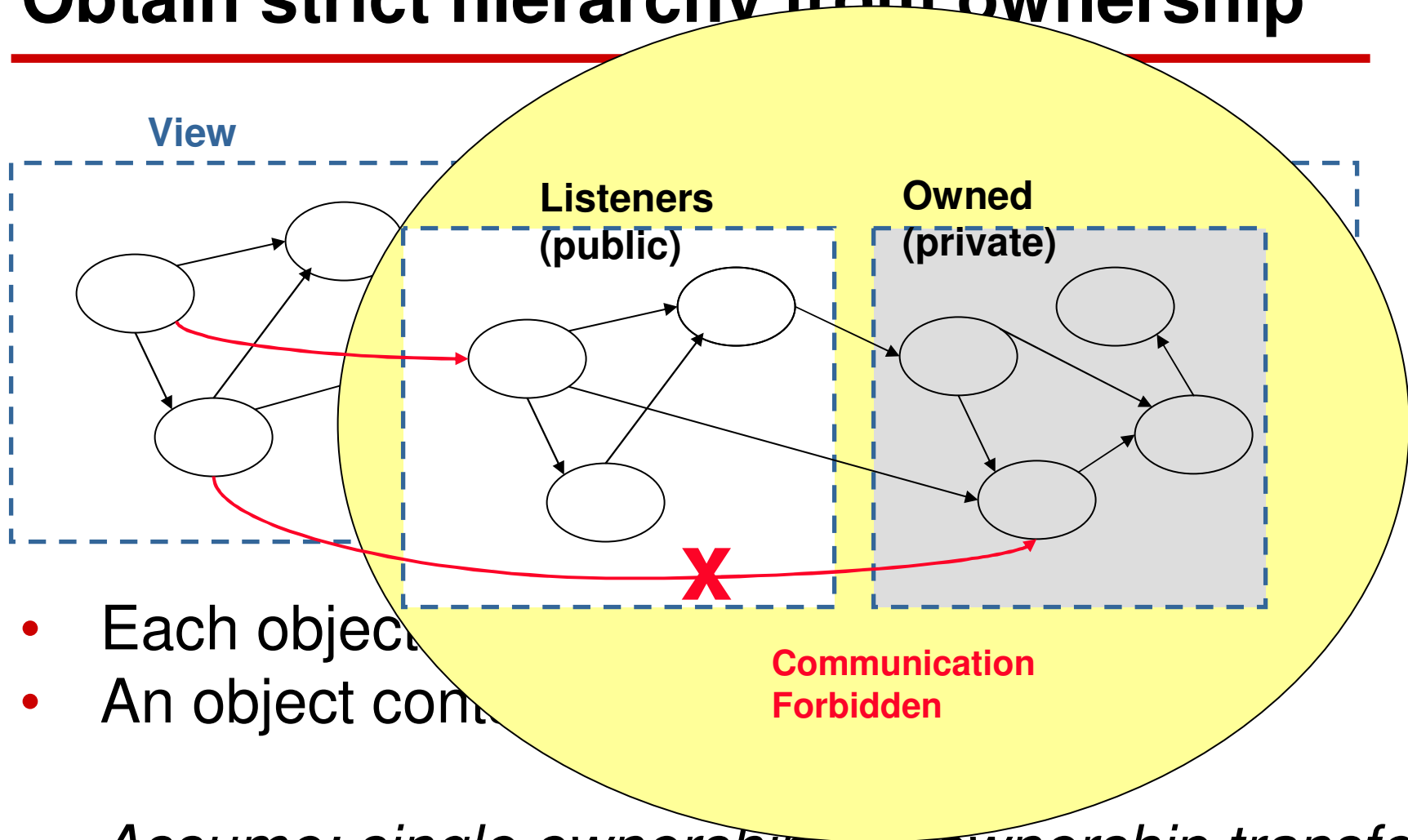
- Nodes represent objects
- Edges represent relations between objects
- Use **ownership types** to obtain information about runtime object structures from compile-time types

Add ownership structure to object graph



- Group objects into *ownership domains*
- Domain names convey design intent
- Precision about inter-domain aliasing
- *Domain links* abstract communication permissions

Obtain strict hierarchy from ownership



- Each object...
- An object contains...
- *Assume: single ownership, no ownership transfer*

Execution Structure based on Ownership

- Introduction
- Approach
- **Details**
- Validation

Example: MiniBank

```
class Branch< CUSTOMERS > {
  domain TELLERS, VAULTS;
  link TELLERS -> VAULTS;

  CUSTOMERS Customer c1;

  TELLERS Teller t1;
  TELLERS Teller t2;

  VAULTS Vault v1;
  VAULTS Vault v2;
}
```

```
class Bank {
  domain owned;

  owned Branch<owned> b1;
}
```

Explanation of annotation syntax:

a B g1: declare object **g1** of type **B** in domain **a**

[public] domain a: declare private [or public] domain

class C < d >: declare formal domain parameter **d** on class **C**

C<owned> c: provide actual domain for formal domain parameter on instance of **C**

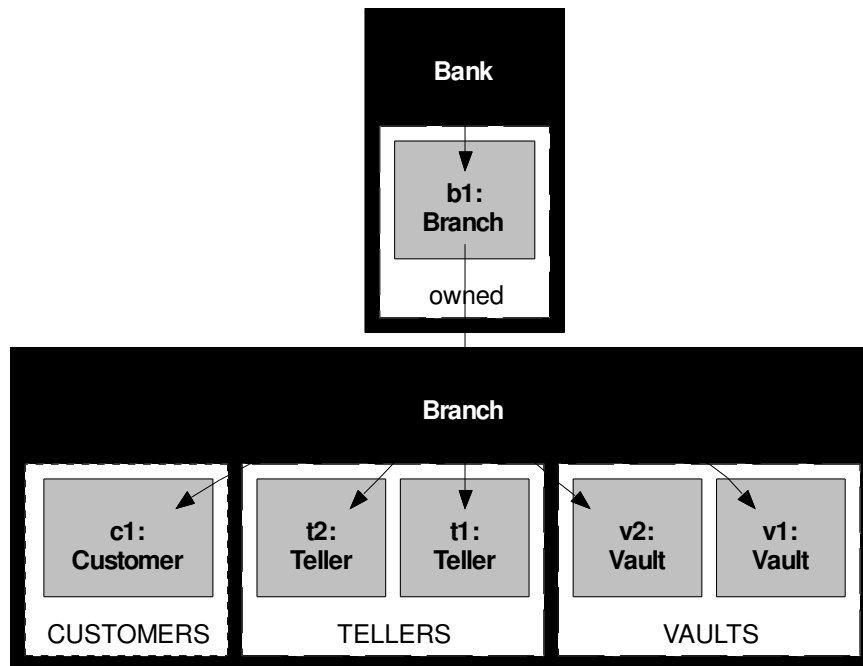
link b -> d: give domain **b** permission to access domain **d**

Intermediate Representations

- Abstract Graph
- Visual Graph
- Ownership Object Graph

MiniBank: Abstract Graph

- Each type has domains declared in it
- Each *abstract domain* shows field and variable declarations as *abstract objects*

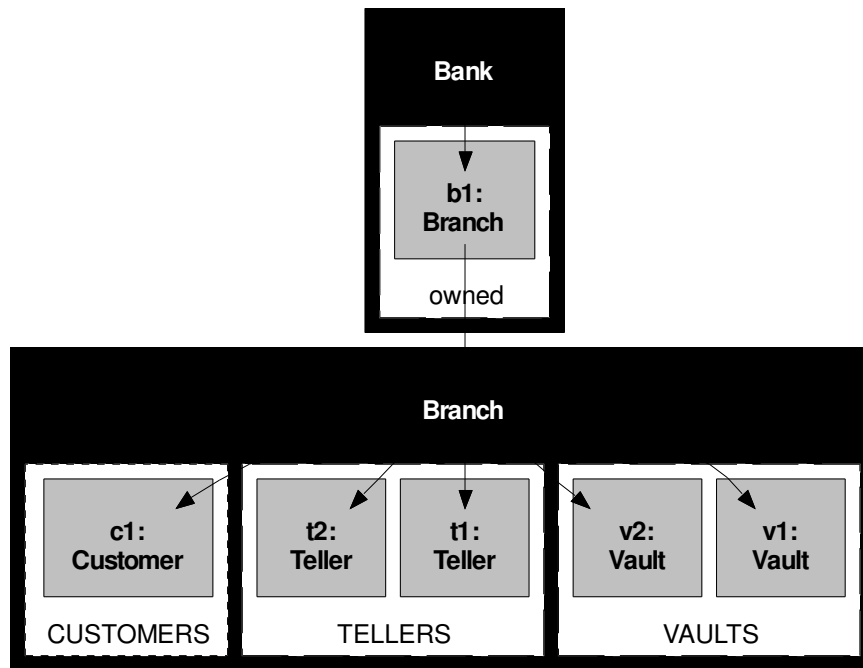


LEGEND

- A **black-filled** box represents a **type**
- White-filled box is *abstract domain* declared in type
- Grey-filled box is *abstract object* declared inside domain

Abstract Graph

- Not suitable as an object graph
- Must show objects with nested domains and objects inside those domains



Abstract Graph Visual Graph

- Visual Graph is the *runtime object graph computed* from the source code
 - This graph is given to dynamic analyses
- Instantiate types in abstract graph
 - *abstract domain* *visual domain*
 - *abstract object* *visual object*

[Domain] The root *visual domain*

[Object] contains *visual objects*, which

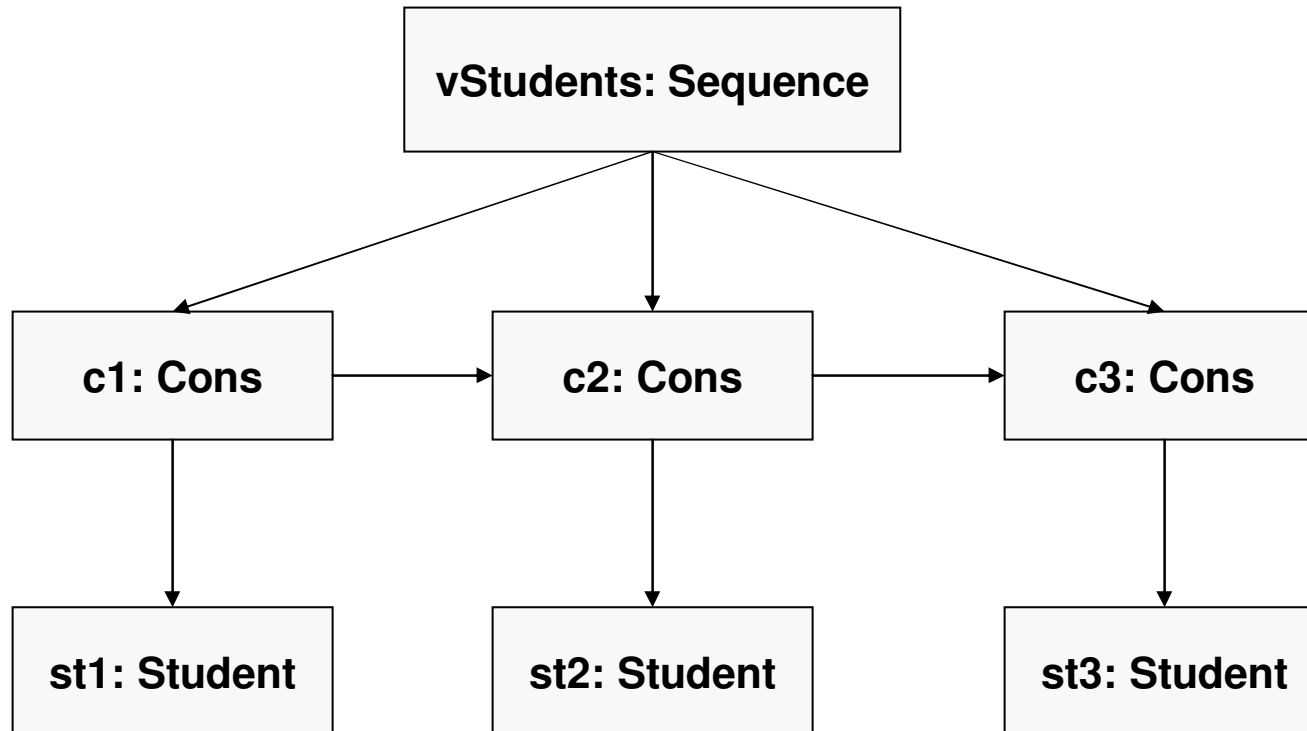
[Domain] in turn, contain *visual domains*,

[Object] ...

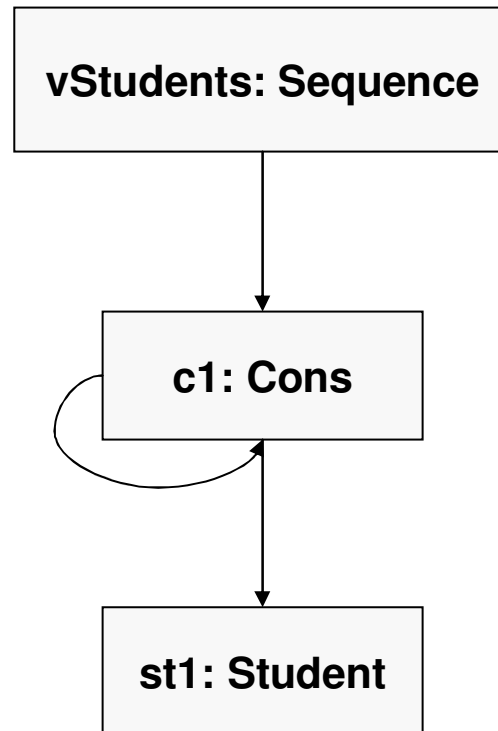
Challenge: How to summarize objects?

- Different executions have different number of objects
- Execution structure must faithfully represent runtime object graph
 - Summarize multiple run-time objects with a canonical object at compile-time
 - Each object in the runtime object graph is represented by exactly one object in the visual graph

From many objects at runtime ...



... to one object at compile-time

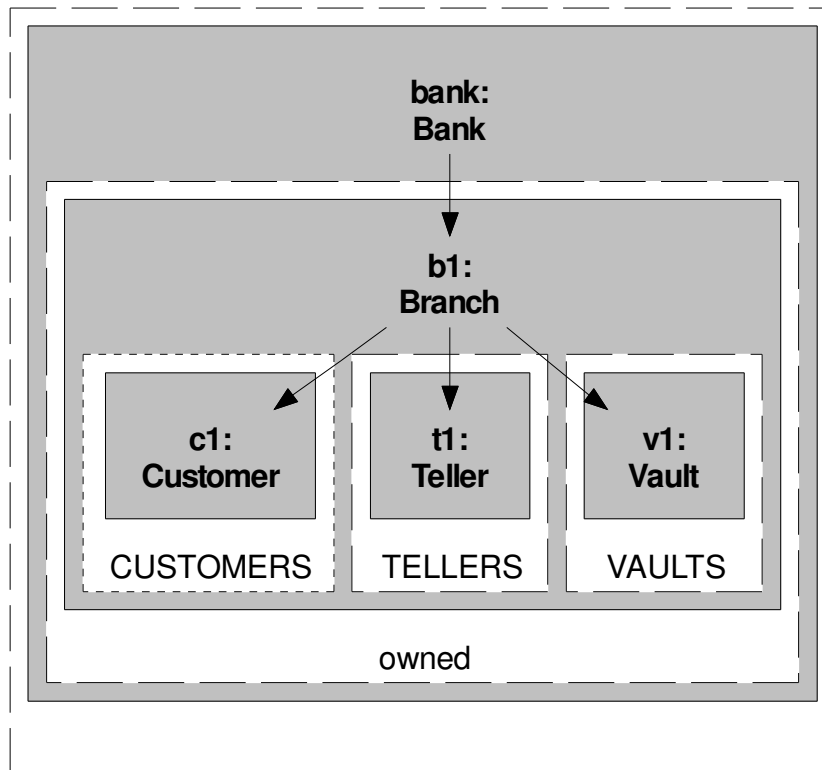


Visual Graph: Merging Objects

- *Invariant: Merge two objects of the “same type” that are in the same domain*
- “Same type” can mean:
 - Same *declared type* or subtype thereof
 - Compatible types (more later)

MiniBank: Merging Objects

- Merge objects of same type in the same domain
 - Objects **t1** and **t2** merged in **TELLERS**
 - Objects **v1** and **v2** merged in **VAULTS**

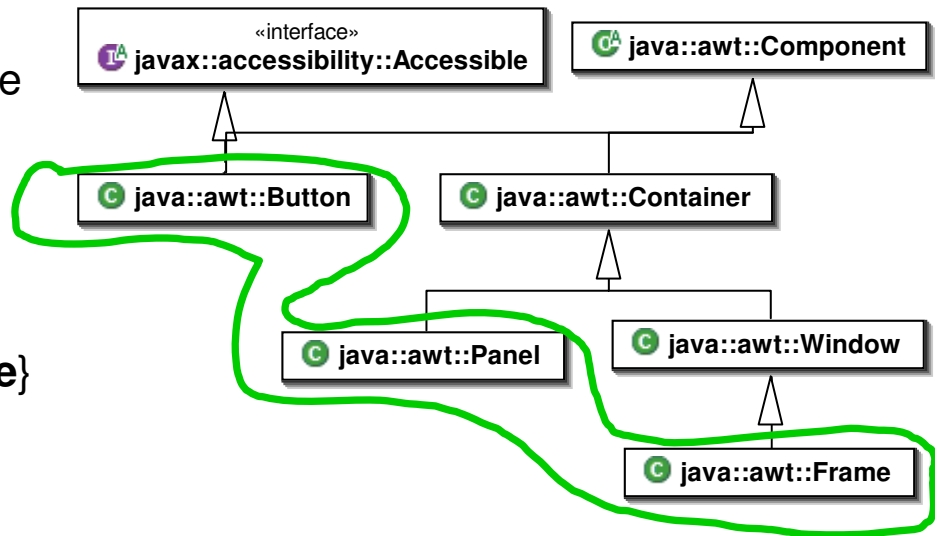


LEGEND

- A white-filled dashed-border box is a **domain**
- Grey-filled box is an **object** in a domain
- Dotted-border shows a **formal** domain
- Object label of the form ***declName : Type***

Visual Graph: Type Abstraction

- **Window** variable and **Frame** variable in the same domain may alias and must be merged for soundness
- **Button**, **Panel** and **Frame** variables in same domain have *least-upper-bound* intersection type {**Component**, **Accessible**} and get heuristically merged



- Merge objects when they share non-trivial least upper bound types
- User configures list of “trivial types”
- Heuristic merging to improve abstraction and reduce clutter in the graph, with soundness

Challenge: How to deal with ownership domain parameters?

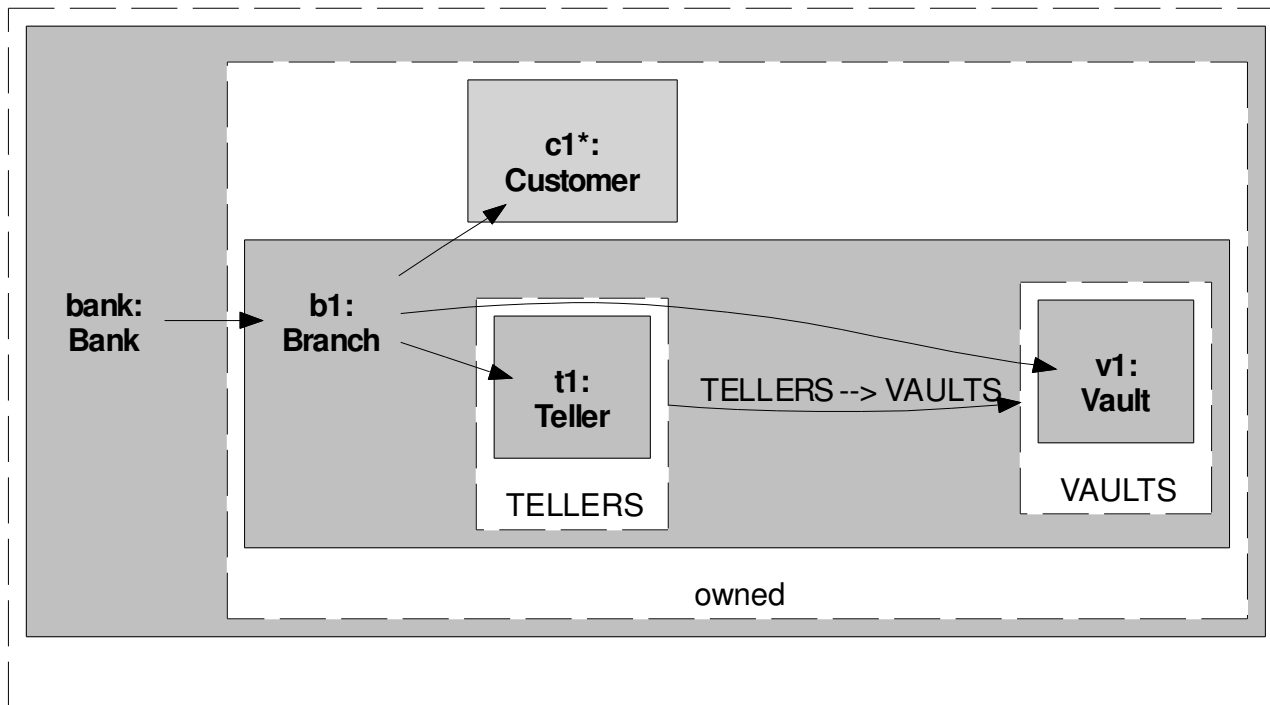
- **Problem:** source code does not directly show what objects are in each domain
- Dynamic analyses need not handle this
 - Parameterization does not exist at runtime

Visual Graph: Pulling Objects

- *Invariant: in visual graph, each runtime object that is actually in a given domain must appear where domain is declared*
 - Take each object *declared* inside a *formal* domain parameter
 - Pull it into each domain that is bound to the formal domain
 - Pull “up” only

MiniBank: Pulling Objects

- Pull object **c1*** from **CUSTOMERS** domain parameter into actual domain **Bank.owned** to which it is bound
- No longer show formal domain parameter



LEGEND

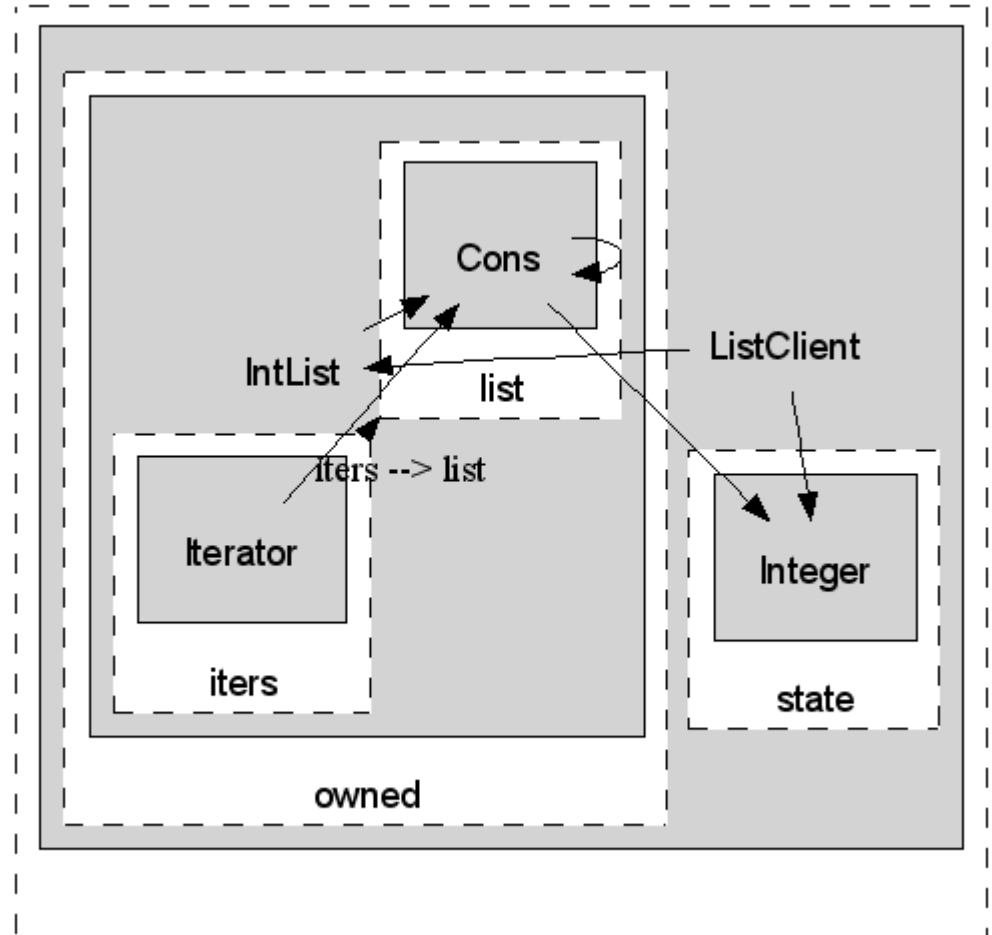
- Solid edge is field reference
- Dashed edge is domain link

Ownership Object Graph

- Visual Graph can be infinite
- Ownership Object Graph
 - Unroll Visual Graph to limited *depth*

Ownership Object Graph: Projecting Visual Graph to limited depth

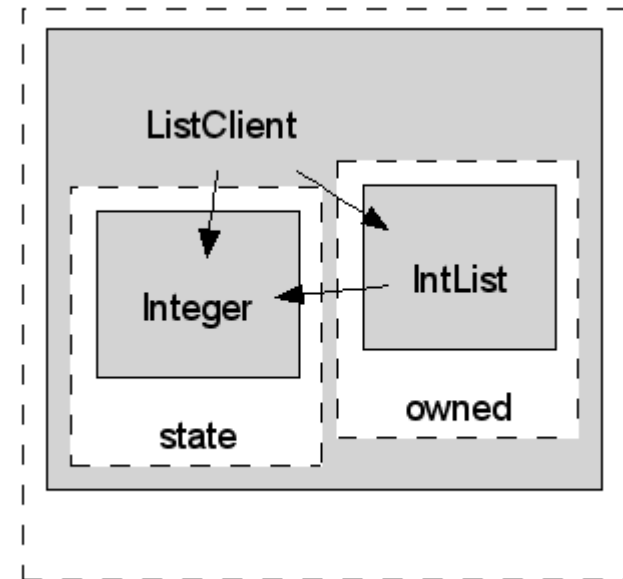
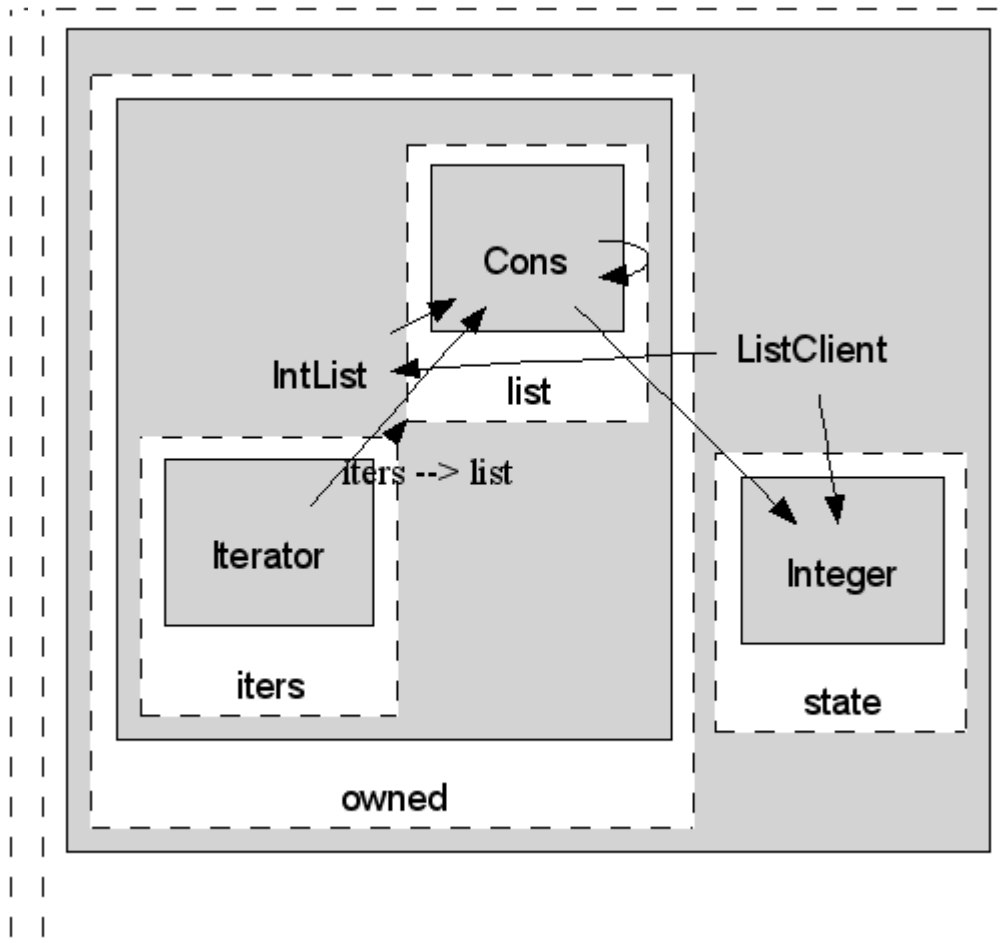
```
class IntList <elems> {  
  domain list;  
  link list -> elems;  
  
  list Cons<elems> head;  
  ...  
}  
  
class Cons <elems> {  
  elems Integer obj;  
  owner Cons<elems> next;  
}  
  
class ListClient {  
  domain owned, state;  
  
  owned IntList<state> list;  
  state Integer i;  
  ...  
}
```



Ownership Object Graph: Adding summary edges

- Visual object summarizes runtime object
- We also need edge summaries
 - Summary edges added to parent objects
 - When sub-objects point to other external objects

Ownership Object Graph: Adding summary edges



Execution Structure based on Ownership

- Introduction
- Approach
- Details

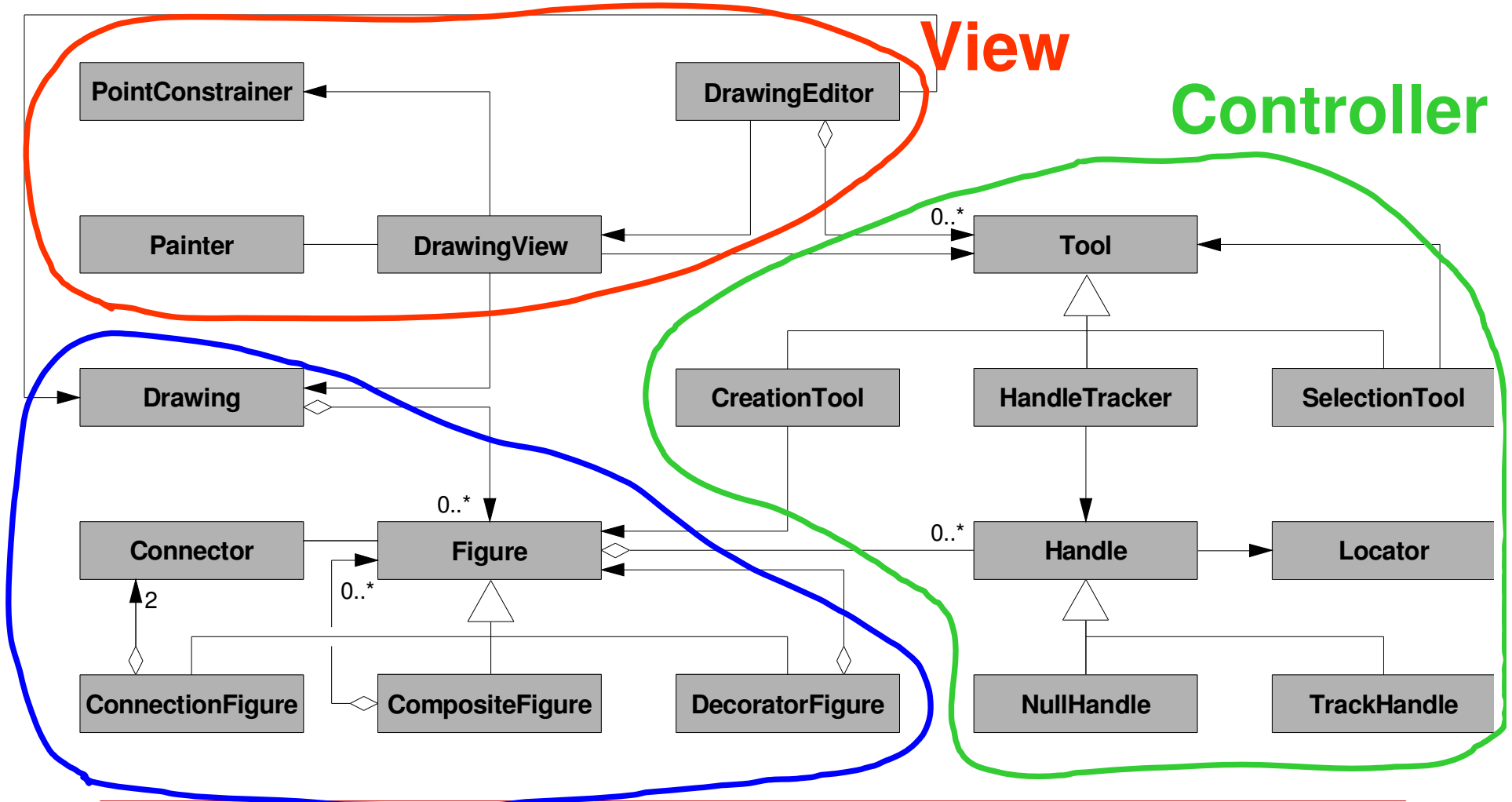
Validation

- Case Study: JHotDraw
- Case Study: HillClimber

Case Study: JHotDraw

- 15,000 lines of Java
- Designed by experts

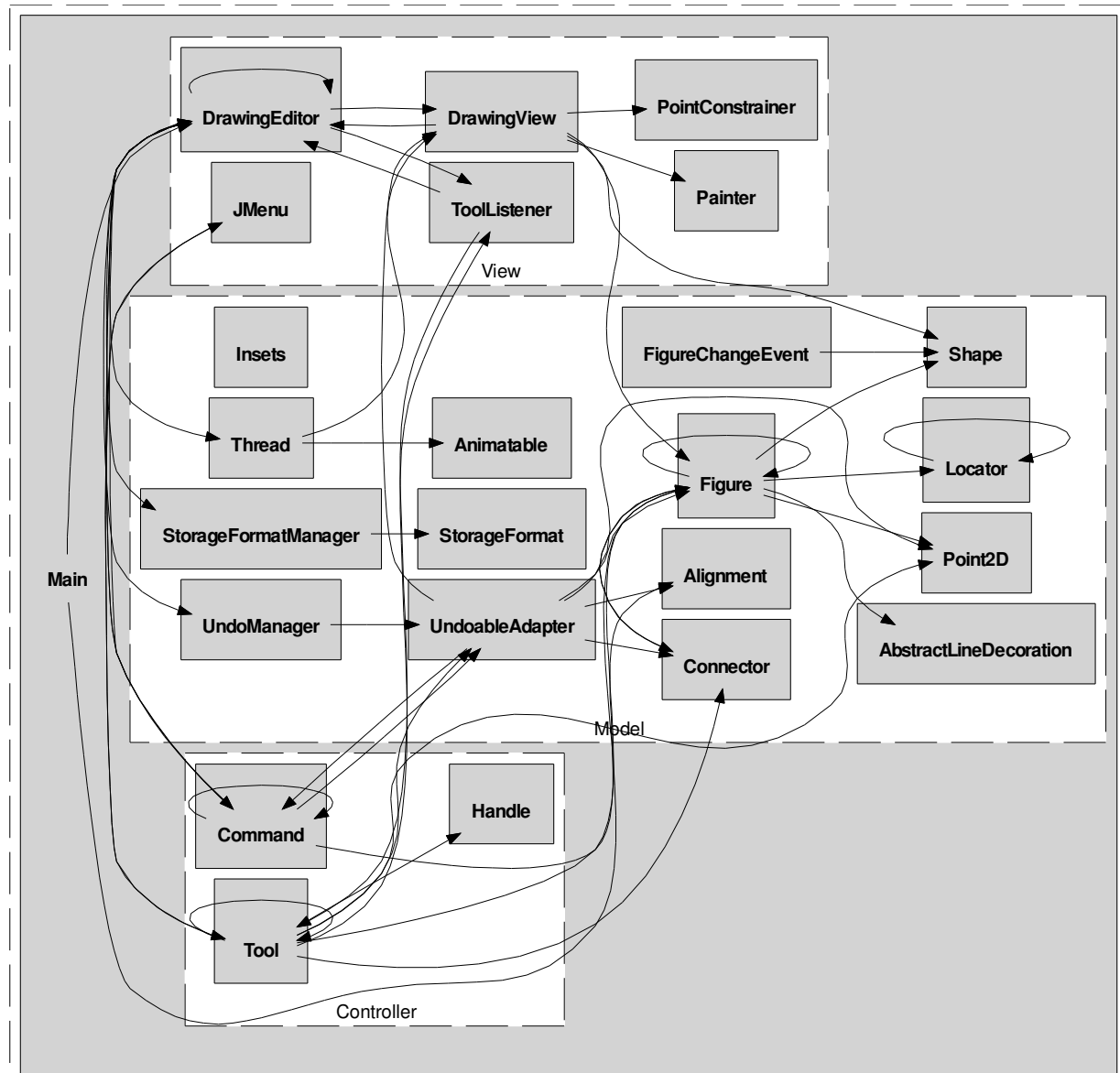
JHotDraw Code Structure



Model

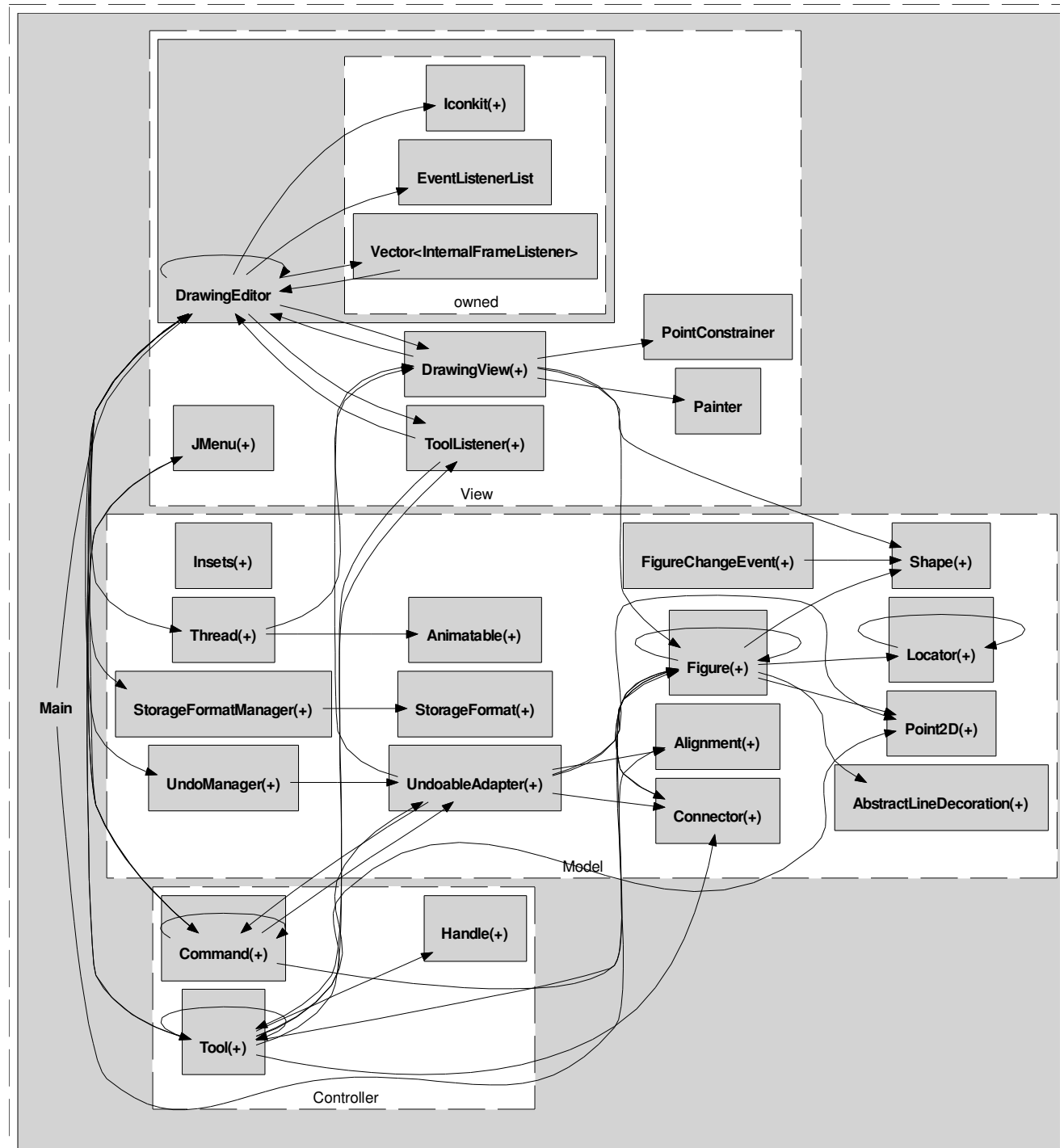
JHotDraw Execution Structure

- Conveys design intent (MVC)
- Fits on one readable page
- Minimum projection depth



JHotDraw

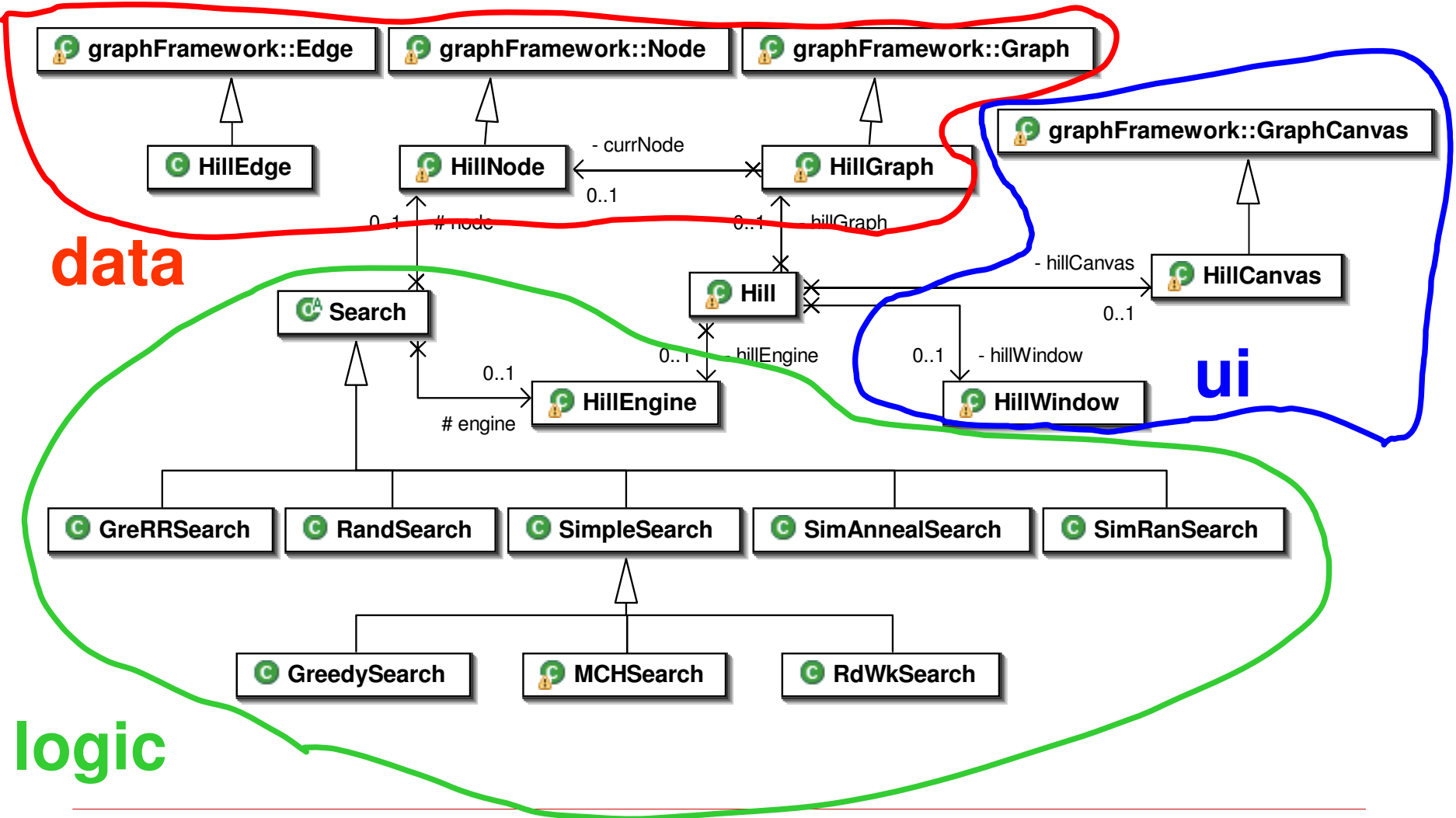
- Increase depth
- Hide internals of individual objects
 - Has (+) on label



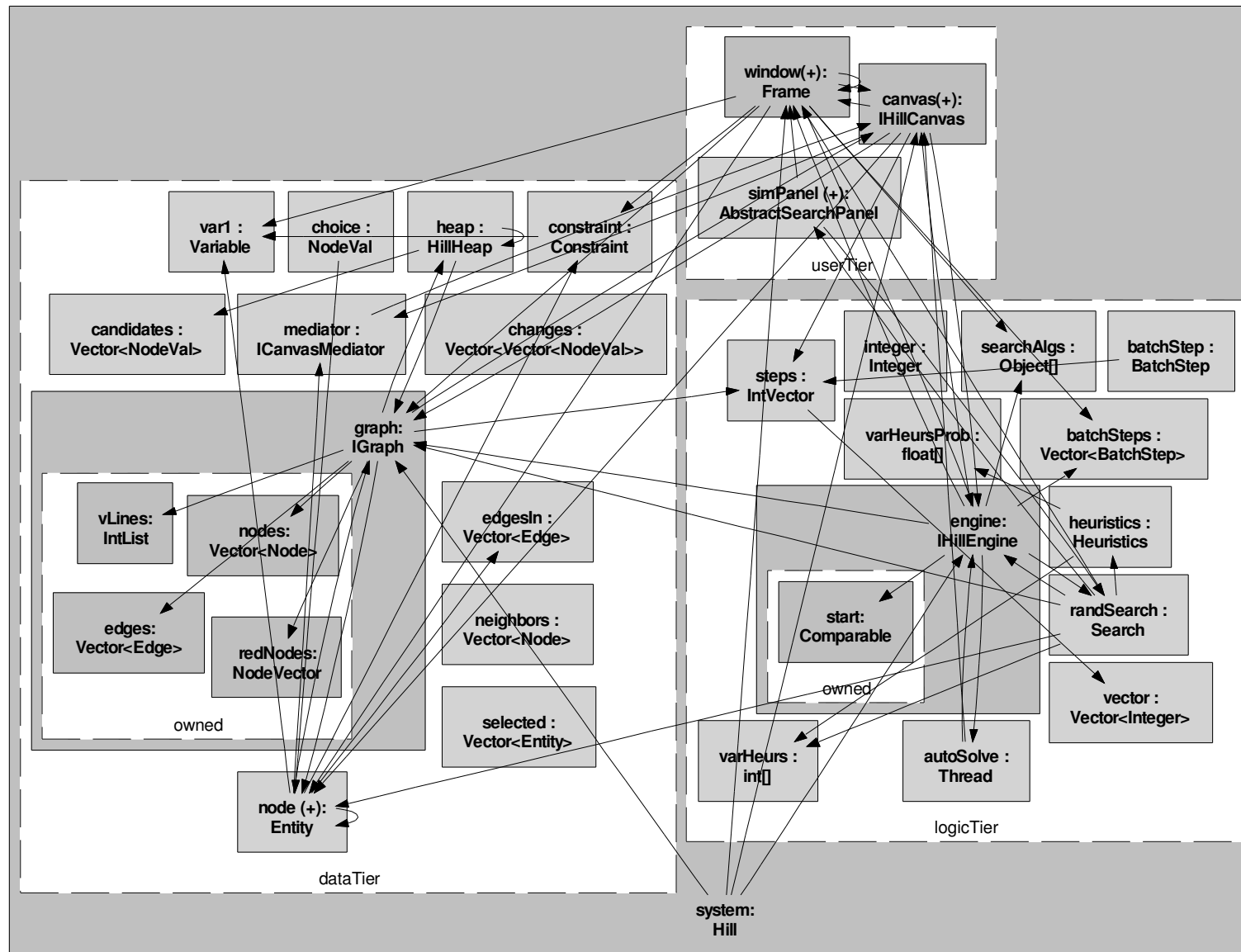
Case Study: HillClimber

- 15,000 lines of Java
- Designed by undergraduates

HillClimber Code Structure



HillClimber Execution Structure



Future Work

- Ownership Object Graph theory:
 - Proof of soundness
- Miscellaneous issues:
 - Flow analysis to resolve 'lent' and 'unique'
 - Add precision (e.g., multiplicities)
- Show Ownership Object Graph useful for reasoning about runtime properties:
 - Performance
 - Distributing an application
 - Security
 - ...

Summary

- Ownership domain annotations enable a compile-time sound execution structure, the Ownership Object Graph
- Ownership Object Graph
 - Hierarchical
 - Conveys design intent
 - More readable than flat raw object graphs obtained without annotations
 - Complements code structure provided by existing tools

For More Information

- “Ownership Domains in the Real World”
 - About annotating JHotDraw and HillClimber
- Tool Demonstration at ECOOP’07
 - *D7 — Eclipse Plug-ins for Statically Checking and Visualizing Ownership Domain Annotations*
 - **Thursday, August 2, 16:45 - 17:30**
 - <http://ecoop07.swt.cs.tu-berlin.de/demos/d7.html>

References

- [AC04] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, 2004.
- [FF06] C. Flanagan and S. N. Freund. Dynamic Architecture Extraction. In *Workshop on Formal Approaches to Testing and Runtime Verification*, 2006.
- [JW01] D. Jackson and A. Waingold. Lightweight Extraction of Object Models from Bytecode. *IEEE Transactions on Software Engineering*, 27(2):156{169, 2001.
- [LR03] P. Lam and M. Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *ECOOP*, 2003.
- [OCa01] R. W. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, Carnegie Mellon University, 2001.
- [HNP02] Hill, T., Noble, J. and Potter, J. Scalable Visualisations of Object-Oriented Systems with Ownership Trees. *Journal of Visual Languages and Computing*, 2002.
- [Spi02] A. Spiegel. *Automatic Distribution of Object-Oriented Programs*. PhD thesis, FU Berlin, 2002.