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

Ownership Domains in the Real World

"Papers have been written enough, let us see systems!"
— Reinhard Wilhelm

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

Motivation for this work

"While many ownership systems look promising on small examples, the question of practical usage for large and complicated applications remains unanswered" – Stefan Nägeli

Hidden Motivation for this Work

How to stop worrying

*about **iterators***

and start living

with real object-oriented code

with apologies to Dale Carnegie

Ownership Domains

- Among many ownership type systems
  - Each object can have one or more domains
  - Each object is in exactly one domain
- Previously implementation
  - Used language extensions
  - Ran on research infrastructure

Ownership Domains with Annotations

- Use Java 1.5 annotations (JSR 175)
- Move to Eclipse infrastructure
- Advantages of using annotations
  - Improved tool support
  - Incrementally annotate large code bases
- Disadvantages
  - Many restrictions imposed by JSR 175
  - Heavy syntactic baggage
  - See details in paper

Annotation Language, Tool Support

- Examples here use simplified syntax
  - See paper for Java 1.5 version
  - Code slightly simplified for presentation

Ownership Domains Case Studies

- Introduction

  Case Studies
- Ownership Domains Expressiveness
- Ownership Domains Challenges

Case Studies: Subject Systems

- JHotDraw
  - 15 KLOC developed by experts
  - Only added annotations
  - Minor refactoring to use annotation system
  - ~ 60 remaining type errors
- HillClimber
  - 15 KLOC developed by undergraduates
  - Annotations helped discover code smells
  - Introduced refactoring to reduce coupling
  - ~ 40 remaining type errors

Overall Annotation Process

- Iterative based on
  - Improved understanding
  - Refactoring the code
  - Visualizing the annotations
- Annotations helped improve code quality
  - Expose tight coupling
  - Guide to reduce coupling
- Visualizing annotations improved them
  - Group related objects
  - Make more objects **owned**

Ownership Domains Case Studies

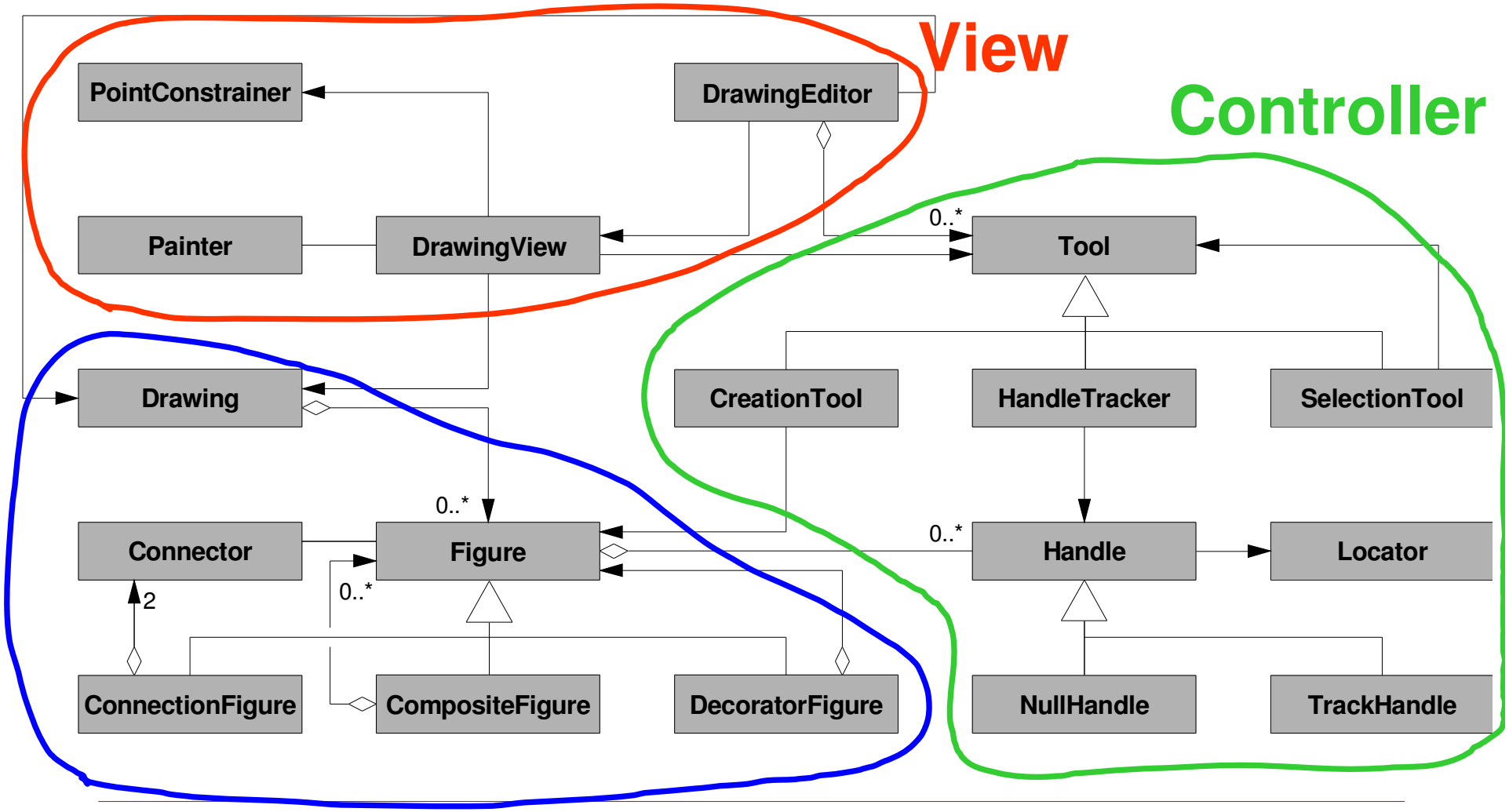
- Introduction
- Case Studies

  Ownership Domains Expressiveness
- Ownership Domains Challenges

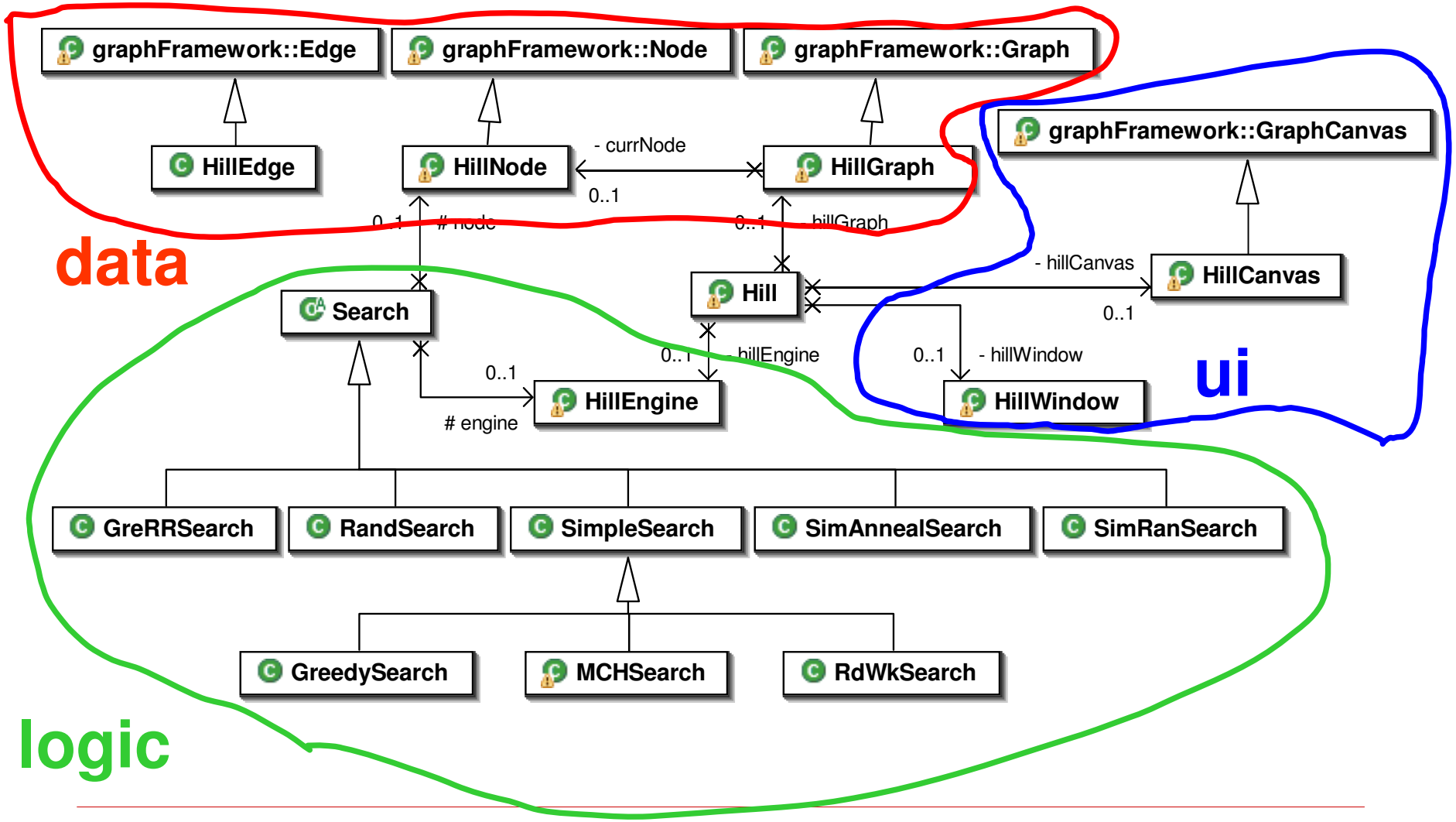
Ownership Domains Expressiveness

- Specify architectural tiers
- Enforce instance encapsulation
- Expose implicit communication
- Expose tight coupling
- Expose and enforce object lifetime
- Promote decoupling code
- Help identify singletons

JHotDraw Ownership Domains



HillClimber Ownership Domains



Specifying architectural tiers

```
class DrawApplication<M,V,C> ... implements DrawingEditor<M,V,C> ... {
}

class MDI_DrawApplication<M,V,C> extends DrawApplication<M,V,C> ... {
}

class JavaDrawApp<M,V,C> extends MDI_DrawApplication<M,V,C> {
}

class Main {
  domains Model, View, Controller;
  ...
  View JavaDrawApp<Model,View,Controller> app = new JavaDrawApp();

  public void run() {
    app.open();
  }

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

Enforcing instance encapsulation

```
/**
 * The interface of a graphical figure. A figure knows its display box
 * and can draw itself. A figure can be composed of several figures.
 * A figure has a set of handles to manipulate its shape or attributes.
 * A figure has one or more connectors that define
 * how to locate a connection point.
*/
interface Figure<M> extends Storable <M> {
 ...
}
```

Enforcing instance encapsulation

```
/**
 * A Figure that is composed of several figures.
 */
abstract class CompositeFigure<M>
          extends AbstractFigure<M> implements FigureChangeListener<M> {

 domain owned;

/**
 * The figures that this figure is composed of
 */
 owned Vector<M Figure<M> > fFigures;

/**
 * Adds a vector of figures.
 */
void addAll(M Vector<M Figure<M>> newFigures) {
  // Cannot assign object M Vector newFigures to owned Vector fFigures
  // this.fFigures = newFigures;

  fFigures.addAll(newFigures);
}
```

Nested parameter (Figure takes 'M' parameter)

Cannot override field marked with 'owned'

Exposing implicit communication

```
/**
 * Drawing is a container for figures. Drawing s
 * events to DrawingChangeListeners whenever a pa
 * invalidated. The Observer pattern is used to decouple the      ing
 * from its views and to enable multiple views.
 */
interface Drawing<M,V> ...{

/**
 * Adds a listener for this drawing.
 * DrawingView implements DrawingChangeListener,
 * so the objects are in 'V' domain paramete
 */
void addDrawingChangeListener(V DrawingChangeListener<M,V> listener);

/**
 * Adds a figure and sets its container to refer to this drawing.
 * @param figure to be added to the drawing
 * @return the figure that was inserted (might be different from the
     figure specified).
 */
 M Figure<M> add(M Figure<M> figure);

}
```

Drawing needs 'V' domain parameter

DrawingChangeListener Implemented by DrawingView (in View domain)

Exposing tight coupling

```
/**
 * Handles are used to change a figure by direct manipulation.
 * Handles know their owning figure and they provide methods to locate
 * the handle on the figure and to track changes.
 * Handles adapt the operations to manipulate a figure to a common
   interface.
*/
interface Handle<M,V,C> {

  /**
   * @deprecated As of version 4.1, use invokeStart(x, y, drawingView)
   */
  void invokeStart(int x, int y, Drawing<M> drawing);

  /**
   * Tracks the start of the interaction.
   * @param x the x position where the interaction
   * @param y the y position where the interaction started
   * @param view the handles container
   */
void invokeStart(int x, int y, V DrawingView<M,V,C> view);

}
```

Assuming Drawing only needs 'M' parameter

DrawingView argument requires all three parameters!

Exposing tight coupling

```
interface Handle<M,V,C> {
  void invokeStart(int x, int y, V DrawingView<M,V,C> view);
  M Undoable<M,V,C> getUndoActivity();
}
```

Assuming Handle only needs 'M' and 'C' parameters

```
interface Handle<M,C> {
  void invokeStart<V> (int x, int y, V DrawingView<M,V,C> view);
  M Undoable<M> getUndoActivity();
}
```

Convert class domain parameter 'V' to method domain parameter

Exposing object lifetime

```
/**
 * AbstractHandle provides default implementation for Handle interface.
 */
abstract class AbstractHandle<M,C> implements Handle<M,C> {

  // The following would not typecheck since 'V' not bound
  V DrawingView<M,V,C> view;

/**
 * @param x the x position where the interaction started
 * @param y the y position where the interaction started
 * @param view the handles container
 */
void invokeStart<V>(int x, int y, V DrawingView<M,V,C> view) {
  // Cannot store argument view in field this.view
  ...
}
```

Convert domain parameter 'V' to method domain parameter

Exposing object lifetime

```
class ResizeHandle<M,V,C> extends LocatorHandle<M,V,C> {

@Override
void invokeStart(int x, int y, V DrawingView<M,V,C> view) {
setUndoActivity(createUndoActivity(view));
...
}

/**
 * Factory method for undo activity. To be overriden by subclasses.
 */
M Undoable<M,V,C> createUndoActivity(V DrawingView<M,V,C> view) {
  unique ResizeHandle.UndoActivity<M,V,C> undoActivity = new
    ResizeHandle.UndoActivity(view);
  return undoActivity;
}

static class UndoActivity<M,V,C> extends UndoableAdapter<M,V,C> {
  UndoActivity(V DrawingView<M,V,C> newView) {
    super(newView);
...}
}

}
```

Exposing object lifetime

```
class UndoableAdapter<M,V,C> implements Undoable<M,V,C> {

 owned Vector<M Figure> myAffectedFigures;

 V DrawingView<M,V,C> myDrawingView;

 UndoableAdapter(V DrawingView<M,V,C> newDrawingView) {
  myDrawingView = newDrawingView;
 }

 void setAffectedFigures(lent FigureEnumeration<M> newAffectedFigures) {
  // the enumeration is not reusable therefore a copy is made
  // to be able to undo-redo the command several time
  rememberFigures(newAffectedFigures);
}

 void rememberFigures(lent FigureEnumeration<M>
   myAffectedFigures = new Vector<Figure>();
   myAffectedFiguresCount = 0;
   while (toBeRemembered.hasMoreElements()) {
    myAffectedFigures.addElement(toBeRemembered.nextElement());
    myAffectedFiguresCount++;
   }
  }
 }
```

Minor violation of MVC design: hold on to the view

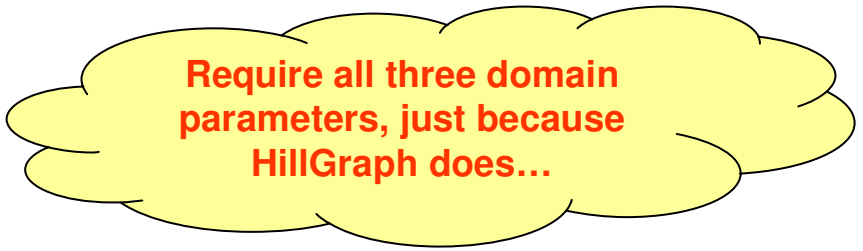
'lent' enforces alias to FigureEnumeration is temporary

Promoting the decoupling of code

- Programming to an interface
- Using the Mediator pattern

When not programming to interface

```
class HillNode<ui,logic,data>  extends Node<data> {

   data HillGraph<ui,logic,data> hillGraph;
}
```



When programming to interface

```
class HillGraph<ui,logic,data> extends Graph<data>
            implements IHillGraph<data> {
}
```

Extract interface IHillGraph that only requires 'data' parameter

```
interface IHillGraph<data>  extends IGraph<data> {
}
```

Program to IHillGraph interface

```
class HillNode<data>  extends Node<data> {
   data IHillGraph<data> hillGraph;
}
```

As a result, HillNode only need 'data' domain parameter

Not using a mediator

```
abstract class Entity<data>  {
  data Graph<data>  graph; // parent graph

...
}

class Node<data>  extends Entity<data>  {
 ...
 int getHeight() {
    return graph.getCanvas().getFontMetrics()...;
  }
}
```

Not using a mediator – bad attempt

```
abstract class Entity<data>  {
  data IGraphCanvas canvas; // 'ui' unbound
...
}

class Node<data>  extends Entity<data>  {
 ...
 int getHeight() {
   return canvas.getFontMetrics()...;
  }
}
```

Defining a mediator

```
/**
 * Mediator interface
 */
interface ICanvasMediator {
  shared FontMetrics getFontMetrics();
}
/**
 * Mediator implementation class
 */
class CanvasMediatorImpl<ui,data> implements ICanvasMediator {

Ui GraphCanvas<ui,data> canvas = null;

CanvasMediatorImpl(ui GraphCanvas<ui,data> canvas) {
 this.canvas = canvas;
}

shared FontMetrics getFontMetrics() {
     return this.canvas.getFontMetrics();
}

}
```

Using a mediator

```
class GraphCanvas<ui,data> extends ... {
 data CanvasMediatorImpl<ui,data> mediator;
 ...
 data ICanvasMediator getMediator() {
   return mediator;
}

abstract class Entity< data> {
   data ICanvasMediator mediator;
 ...
}
class Node<data> extends Entity<data> {
 ...
  /**
   * Gets the height of this node.
   *
   */
  protected int getHeight() {
    return mediator.getFontMetrics().getHeight() + ...;
  }
}
```

Identifying singletons

```
class Iconkit {
 static unique Iconkit fgIconkit = null;

 /**
  * Constructs an Iconkit that uses the given editor
  * to resolve image path names.
  */
 unique
 public Iconkit(unique Component component) {
  ...
  fgIconkit = this;
 }

 /**
  * Gets the single instance
  */
 public unique static Iconkit instance() {
   return fgIconkit;
  }
}
```

Ownership Domains Case Studies

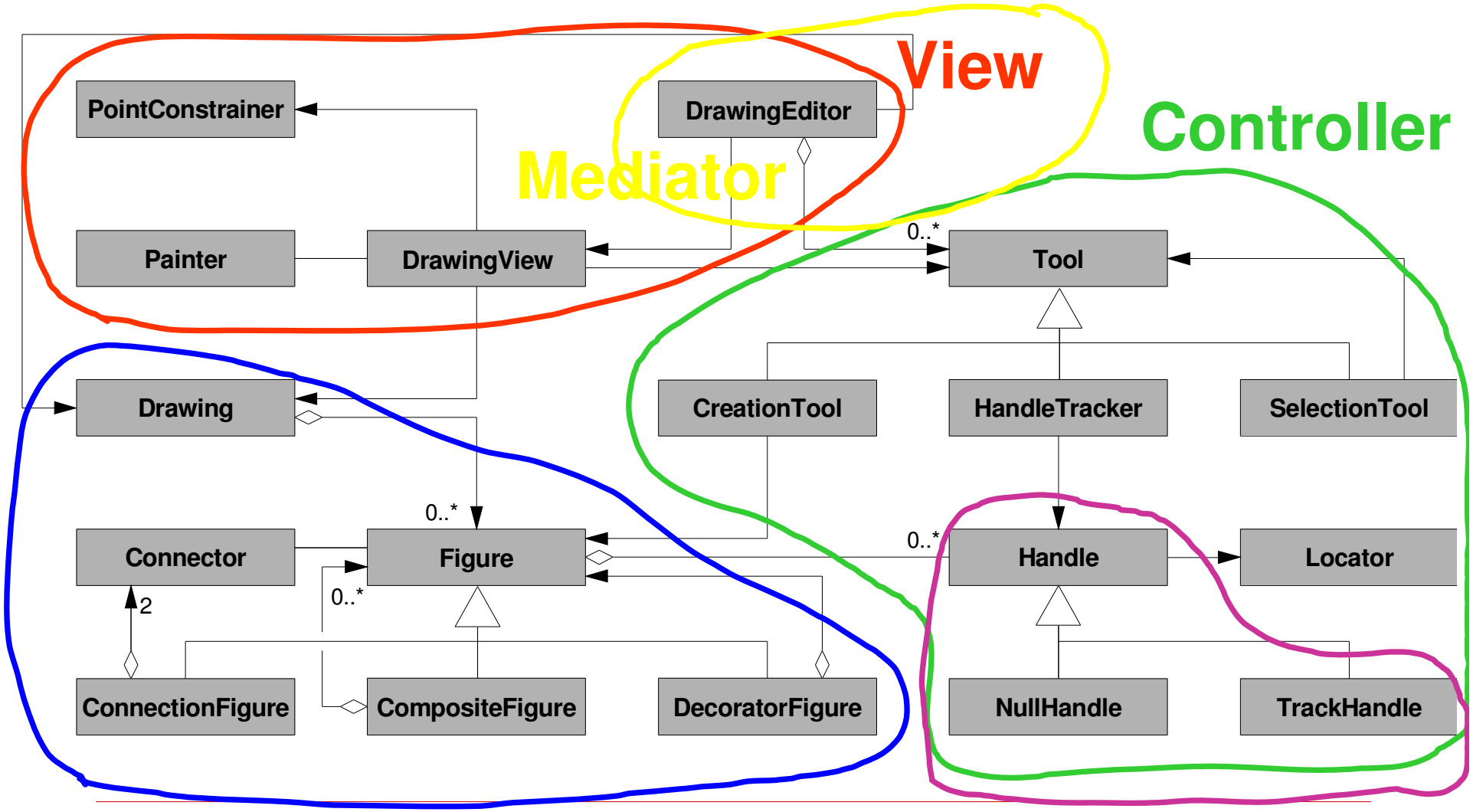
- Introduction
- Case Studies
- Ownership Domains Expressiveness

Ownership Domains Challenges

Ownership Domains Challenges

- Ownership domains vs. ownership intent
- Fake class to declare top-level domains
- One object vs. two conceptual objects
- Annotating listener objects
- Annotating static code
- Having non-verbose annotations

Ownership domains may not correspond to the “ownership design intent"

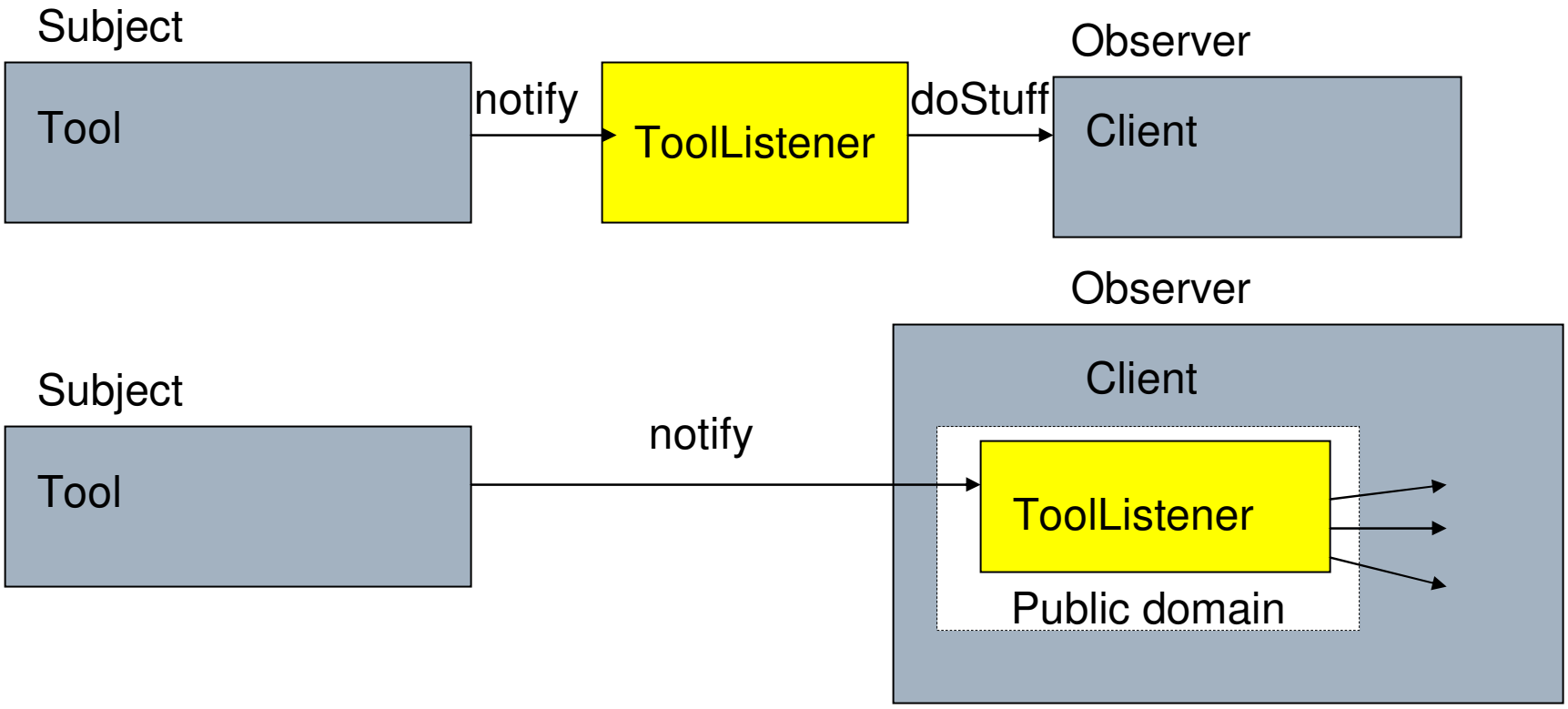


Fake class to declare top-level ownership domains

```
class DrawApplication<M,V,C> ... implements DrawingEditor<M,V,C> ... {
}

class MDI_DrawApplication<M,V,C> extends DrawApplication<M,V,C> ... {
}

class JavaDrawApp<M,V,C> extends MDI_DrawApplication<M,V,C> {
}

class Main {
  domains Model, View, Controller;

  View JavaDrawApp<Model,View,Controller> app = new JavaDrawApp();

  public void run() {
    app.open();
  }

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

Public domains

- Add expressiveness to type system
- Ideal for visualization (grouping objects)



One object vs. two conceptual objects

- Public domains more suitable for composition than inheritance
- Object cannot be “split” into two domains
- Leads to iterative annotation process

```
abstract class AbstractCommand<M,V,C>
        implements Command<M,V,C>,
                   FigureSelectionListener<M,V,C>,
                   ... {

 public domain FIGURESELECTIONLISTENER;

...
}
```

Command cannot split the “listener” part of itself into a public domain

Annotating listener objects

```
class StandardDrawingView<M,V,C> implements DrawingView<M,V,C>, ... {

 /**
  * The registered list of listeners for selection changes
  */
 owned Vector<C FigureSelectionListener<M,V,C>> fSelectionListeners;

 StandardDrawingView(V DrawingEditor<M,V,C> editor, ...) {
    ...
    // DrawingEditor implements FigureSelectionListener
    // editor is in 'V' domain parameter, not 'C'!
    addFigureSelectionListener(editor);
}

 /**
  * Add a listener for selection changes. AbstractCommand implements
  * FigureSelectionListener. Command is in the 'C' domain parameter!
  */
 void addFigureSelectionListener(C FigureSelectionListener<M,V,C> fsl) {
  fSelectionListeners.add(fsl);
  }
}
```

Annotating listener objects – a solution

```
class StandardDrawingView<M,V,C> implements DrawingView<M,V,C>, ... {

 /**
  * The registered list of listeners for selection changes
  */
 owned Vector<? FigureSelectionListener<M,V,C>> fSelectionListeners;

 StandardDrawingView(V DrawingEditor<M,V,C> editor, ...) {
   ...
   // DrawingEditor implements FigureSelectionListener
   // editor is in 'V' domain parameter, not 'C'!
   addFigureSelectionListener(editor);
}

 /**
  * Add a listener for selection changes. AbstractCommand implements
  * FigureSelectionListener. Command is in the 'C' domain parameter!
  */
 void addFigureSelectionListener(? FigureSelectionListener<M,V,C> fsl) {
  fSelectionListeners.add(fsl);
  }
}
```

Static code can be challengin

Hashtable has 3 parameters: key, value, entry.

```
class NullDrawingView<M,V,C> ... implements DrawingView<M,V,C> {

    static @Domain("unique<unique<?,?,?>,unique<?,?,?>,unique>")
    Hashtable<DrawingEditor, DrawingView> dvMgr = new ...
    ...

public synchronized static
   Vx DrawingView<Mx,Vx,Cx>
  getManagedDrawingView<Mx,Vx,Cx> (V1 DrawingEditor<Mx,Vx,Cx> editor) {
    if (dvMgr.containsKey(editor)) {
      Vx DrawingView<Mx,Vx,Cx> drawingView = dvMgr.get(editor);
      return drawingView;
    }
    else {
      Vx DrawingView<Mx,Vx,Cx>newDrawingView=new NullDrawingView(editor);
      dvMgr.put(editor, newDrawingView);
      return newDrawingView;
    }
  }
}
```

Does not typecheck!

Domain parameters declared on class not in scope for static members

Annotations can be verbose

```
class UndoManager<M,V,C> {
  /**
   * Collection of undo activities
   */
  owned Vector<M Undoable<M,V,C>> undoStack;

  void clearStackVerbose(lent Vector<M Undoable<M,V,C>> s) {
    s.removeAllElements();
  }

  void clearStackAny(lent Vector<? Undoable<?,?,?>> s) {
    s.removeAllElements();
  }

  void clearStack(lent Vector<Undoable> s) {
    s.removeAllElements();
  }
}
```

Current annotations needed too verbose

The equivalent of “raw type” on generic type

Related Work

- Earlier HillClimber case study
  - ArchJava + AliasJava language extensions
- Case study by Hächler
  - Evaluated Universes type system
  - Annotate parts of 50,000 LOC system
  - No automated visualization support
- Case study by Nägeli
  - Universes and Ownership Domains
  - Studied design patterns in isolation

Summary

- Re-implementation in Java 1.5
  - Tool support for substantial case studies
  - Access to refactoring tool support crucial
- Two case studies
  - Evaluated ownership domains on real code
  - Identified some interesting outcomes
- Future work
  - Address expressiveness challenges
  - “There is still a lot of road to cover” (reviewer)