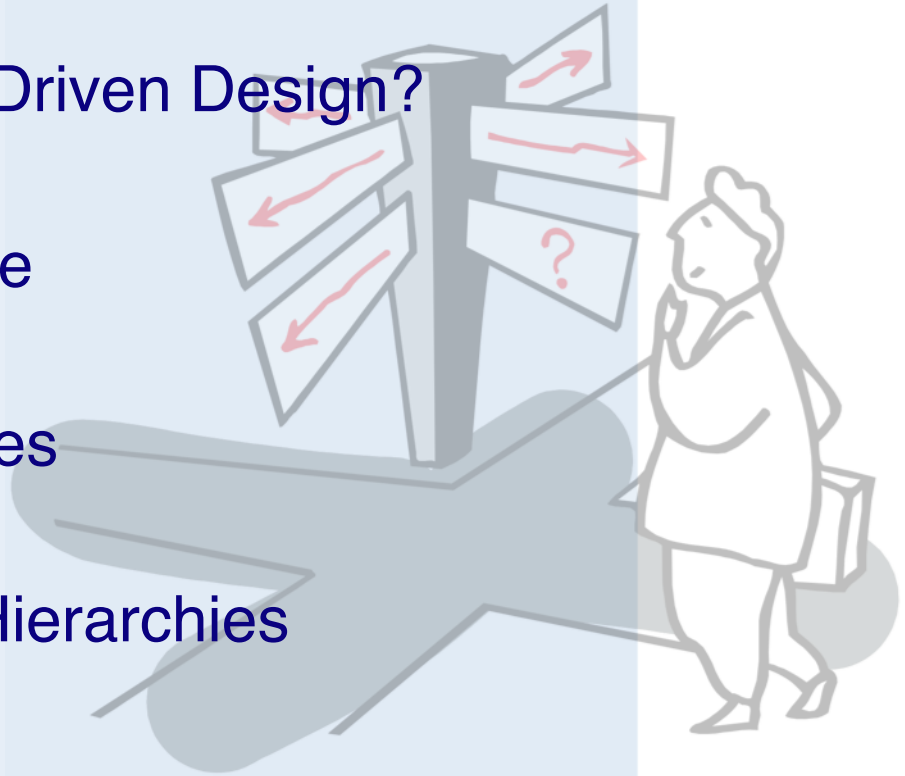
UNIVERSITÄT
BERN

Introduction to Software Engineering

4. Responsibility-Driven Design

Roadmap

- Why use Responsibility-Driven Design?
- Finding Classes
- Class Selection Rationale
- CRC sessions
- Identifying Responsibilities
- Finding Collaborations
- Structuring Inheritance Hierarchies

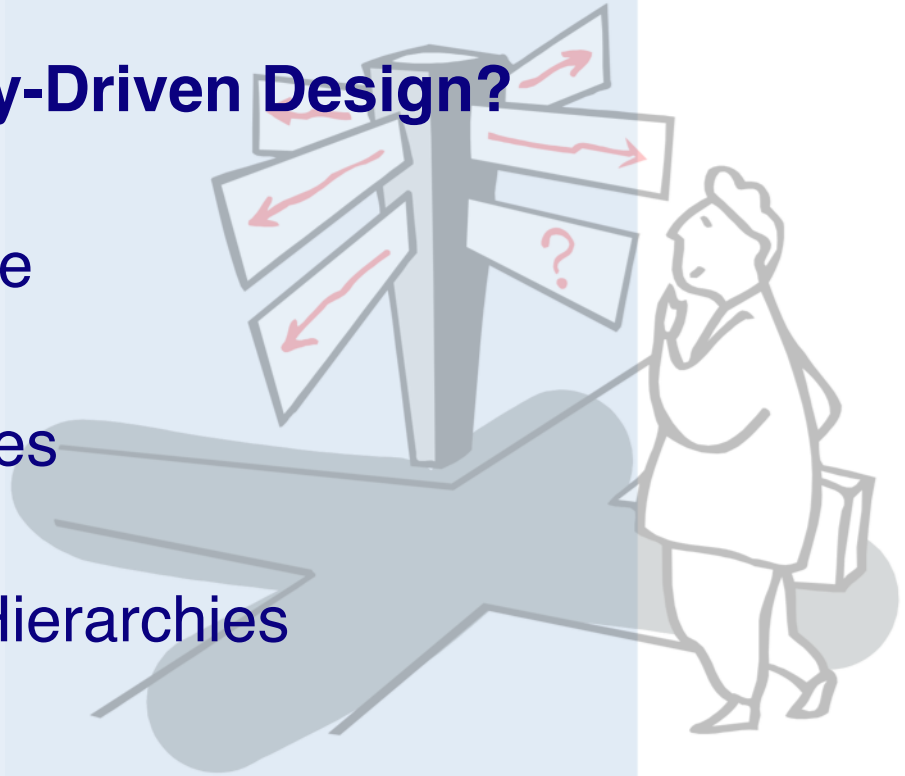


Source

- *Designing Object-Oriented Software*, R. Wirfs-Brock, B. Wilkerson, L. Wiener, Prentice Hall, 1990.

Roadmap

- **Why use Responsibility-Driven Design?**
- Finding Classes
- Class Selection Rationale
- CRC sessions
- Identifying Responsibilities
- Finding Collaborations
- Structuring Inheritance Hierarchies



Why Responsibility-driven Design?

Functional Decomposition:

> *Decompose according to the functions a system is supposed to perform.*

- Good in a “waterfall” approach: stable requirements and one monolithic function

However

- Naive: Modern systems perform more than one function
- Maintainability: system functions evolve ⇒ redesign affect whole system
- Interoperability: interfacing with other system is difficult

Why Responsibility-driven Design?

Object-Oriented Decomposition:

> *Decompose according to the objects a system is supposed to manipulate.*

— Better for complex and evolving systems

However

— How to find the objects?

Iteration in Object-Oriented Design

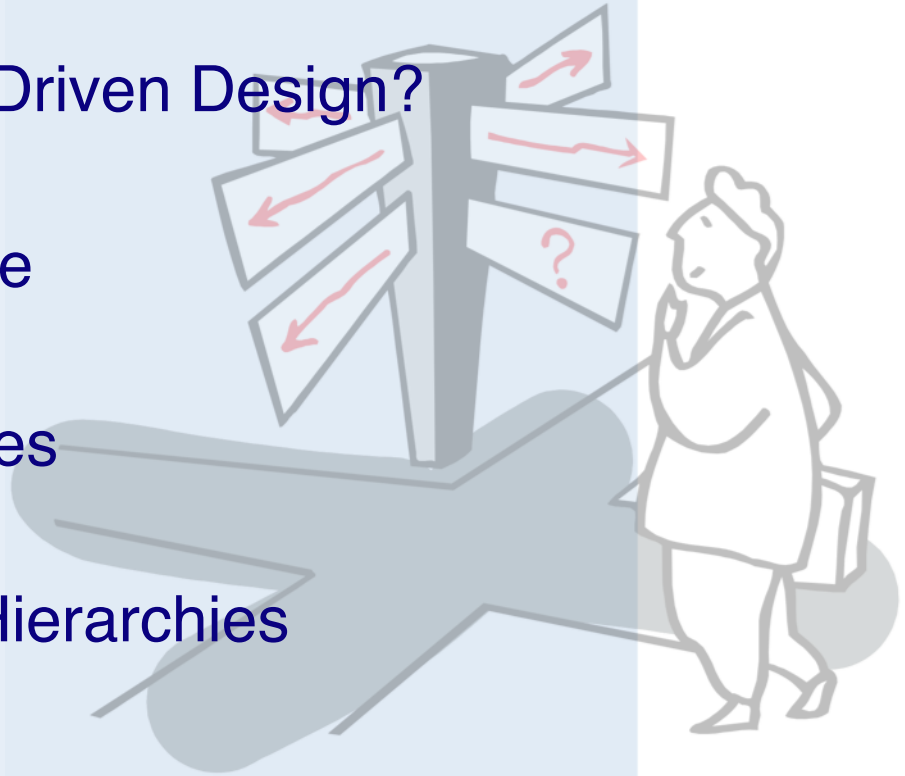
- The result of the design process is *not a final product:*
  - design decisions may be *revisited*, even after implementation
  - design is not linear but *iterative*

- The design process is *not algorithmic*:
  - a design method provides *guidelines*, not fixed rules
  - "a good *sense of style* often helps produce clean, elegant designs — designs that make a lot of sense from the engineering standpoint"

Responsibility-driven design is an (analysis and) design technique that works well in combination with various methods and notations.

Roadmap

- Why use Responsibility-Driven Design?
- **Finding Classes**
- Class Selection Rationale
- CRC sessions
- Identifying Responsibilities
- Finding Collaborations
- Structuring Inheritance Hierarchies



The Initial Exploration

1. Find the *classes* in your system

2. Determine the *responsibilities* of each class

3. Determine how objects *collaborate* with each other to fulfil their responsibilities

The Detailed Analysis

1. *Factor* common responsibilities to build class hierarchies
2. *Streamline* collaborations between objects
   - Is message traffic heavy in parts of the system?
   - Are there classes that collaborate with everybody?
   - Are there classes that collaborate with nobody?
   - Are there groups of classes that can be seen as subsystems?
3. Turn class responsibilities into fully specified signatures

Finding Classes

Start with requirements specification:

What are the goals of the system being designed, its expected inputs and desired responses?

1. Look for *noun phrases*:
   - separate into obvious classes, uncertain candidates, and nonsense

Finding Classes ...

2. Refine to a list of *candidate classes*. Some guidelines are:

   - Model *physical objects* — e.g. disks, printers
   - Model *conceptual entities* — e.g. windows, files
   - Choose *one word for one concept* — what does it mean within the system
   - Be wary of *adjectives* — is it really a separate class?
   - Be wary of *missing or misleading subjects* — rephrase in active voice
   - Model *categories of classes* — delay modelling of inheritance
   - Model *interfaces* to the system — e.g., user interface, program interfaces
   - Model attribute *values*, not attributes — e.g., Point vs. Centre

Drawing Editor Requirements Specification

The drawing editor is an interactive graphics editor. With it, users can create and edit drawings composed of lines, rectangles, ellipses and text.

Tools control the mode of operation of the editor. Exactly one tool is active at any given time.

Two kinds of tools exist: the selection tool and creation tools. When the selection tool is active, existing drawing elements can be selected with the cursor. One or more drawing elements can be selected and manipulated; if several drawing elements are selected, they can be manipulated as if they were a single element. Elements that have been selected in this way are referred to as the current selection. The current selection is indicated visually by displaying the control points for the element. Clicking on and dragging a control point modifies the element with which the control point is associated.

When a creation tool is active, the current selection is empty. The cursor changes in different ways according to the specific creation tool, and the user can create an element of the selected kind. After the element is created, the selection tool is made active and the newly created element becomes the current selection.

The text creation tool changes the shape of the cursor to that of an I-beam. The position of the first character of text is determined by where the user clicks the mouse button. The creation tool is no longer active when the user clicks the mouse button outside the text element. The control points for a text element are the four corners of the region within which the text is formatted. Dragging the control points changes this region. The other creation tools allow the creation of lines, rectangles and ellipses. They change the shape of the cursor to that of a crosshair. The appropriate element starts to be created when the mouse button is pressed, and is completed when the mouse button is released. These two events create the start point and the stop point.

The line creation tool creates a line from the start point to the stop point. These are the control points of a line. Dragging a control point changes the end point.

The rectangle creation tool creates a rectangle such that these points are diagonally opposite corners. These points and the other corners are the control points. Dragging a control point changes the associated corner.

The ellipse creation tool creates an ellipse fitting within the rectangle defined by the two points described above. The major radius is one half the width of the rectangle, and the minor radius is one half the height of the rectangle. The control points are at the corners of the bounding rectangle. Dragging control points changes the associated corner.

Drawing Editor: noun phrases

The drawing editor is an interactive graphics editor. With it, users can create and edit drawings composed of lines, rectangles, ellipses and text.

Tools control the mode of operation of the editor. Exactly one tool is active at any given time.

Two kinds of tools exist: the selection tool and creation tools. When the selection tool is active, existing drawing elements can be selected with the cursor. One or more drawing elements can be selected and manipulated; if several drawing elements are selected, they can be manipulated as if they were a single element. Elements that have been selected in this way are referred to as the current selection. The current selection is indicated visually by displaying the control points for the element. Clicking on and dragging a control point modifies the element with which the control point is associated.

When a creation tool is active, the current selection is empty. The cursor changes in different ways according to the specific creation tool, and the user can create an element of the selected kind. After the element is created, the selection tool is made active and the newly created element becomes the current selection.

…

The text creation tool changes the shape of the cursor to that of an I-beam. The position of the first character of text is determined by where the user clicks the mouse button. The creation tool is no longer active when the user clicks the mouse button outside the text element. The control points for a text element are the four corners of the region within which the text is formatted. Dragging the control points changes this region. The other creation tools allow the creation of lines, rectangles and ellipses. They change the shape of the cursor to that of a crosshair. The appropriate element starts to be created when the mouse button is pressed, and is completed when the mouse button is released. These two events create the start point and the stop point.

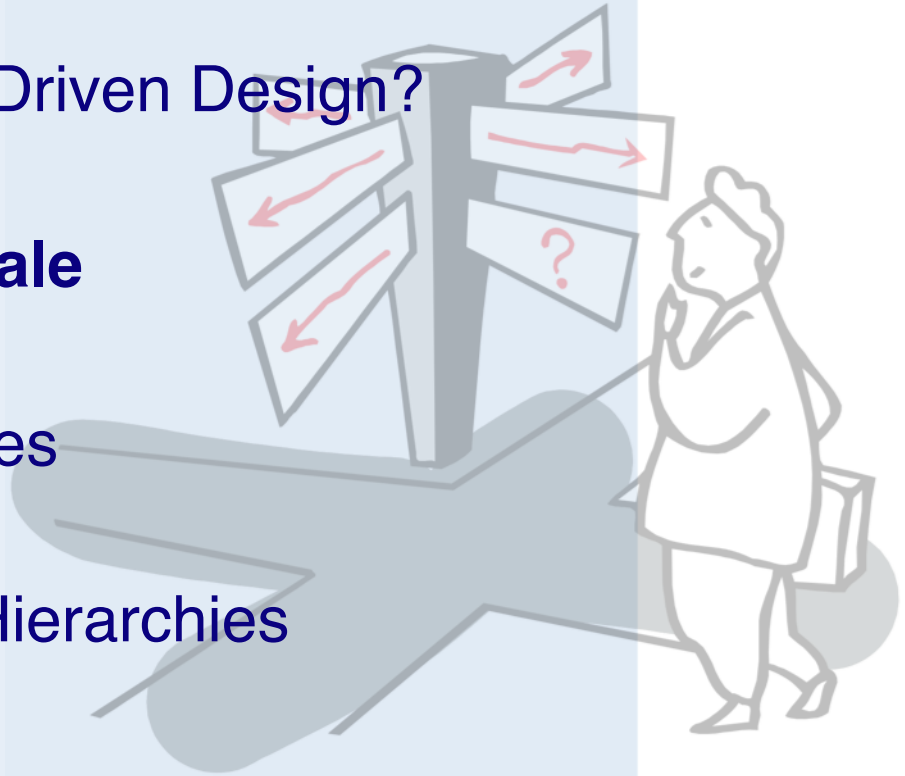
The line creation tool creates a line from the start point to the stop point. These are the control points of a line. Dragging a control point changes the end point.

The rectangle creation tool creates a rectangle such that these points are diagonally opposite corners. These points and the other corners are the control points. Dragging a control point changes the associated corner.

The ellipse creation tool creates an ellipse fitting within the rectangle defined by the two points described above. The major radius is one half the width of the rectangle, and the minor radius is one half the height of the rectangle. The control points are at the corners of the bounding rectangle. Dragging control points changes the associated corner.

Roadmap

- Why use Responsibility-Driven Design?
- Finding Classes
- **Class Selection Rationale**
- CRC sessions
- Identifying Responsibilities
- Finding Collaborations
- Structuring Inheritance Hierarchies



Class Selection Rationale

Model physical objects:

- ~~mouse button~~ [event or attribute]

Model conceptual entities:

- ellipse, line, rectangle
- Drawing, Drawing Element
- Tool, Creation Tool, Ellipse Creation Tool, Line Creation Tool, Rectangle Creation Tool, Selection Tool, Text Creation Tool
- text, Character
- Current Selection

Class Selection Rationale ...

Choose one word for one concept:

- Drawing Editor ⇒
  ~~editor, interactive graphics editor~~
- Drawing Element ⇒ ~~element~~
- Text Element ⇒ ~~text~~
- Ellipse Element, Line Element, Rectangle Element
  ⇒ ~~ellipse, line, rectangle~~

Class Selection Rationale ...

Be wary of adjectives:

- Ellipse Creation Tool, Line Creation Tool, Rectangle Creation Tool, Selection Tool, Text Creation Tool
  - all have different requirements
- ~~bounding rectangle, rectangle, region~~ ⇒ Rectangle
  - common meaning, but different from Rectangle Element
- Point ⇒ ~~end point, start point, stop point~~
- Control Point
  - more than just a coordinate
- corner ⇒ ~~associated corner, diagonally opposite corner~~
  - no new behaviour

Class Selection Rationale ...

Be wary of sentences with missing or misleading subjects:

- "The current selection is indicated visually by displaying the control points for the element."
  - by what? Assume Drawing Editor ...

Model categories:

- Tool, Creation Tool

Model interfaces to the system: — no good candidates here ...

- ~~user~~ — *don't need to model user explicitly*
- ~~cursor~~ — *cursor motion handled by operating system*

Class Selection Rationale ...

Model values of attributes, not attributes themselves:

- ~~height of the rectangle, width of the rectangle~~
- ~~major radius, minor radius~~
- ~~position~~ *— of first text character; probably Point attribute*
- ~~mode of operation~~ *— attribute of Drawing Editor*
- ~~shape of the cursor, I-beam, crosshair~~ *— attributes of Cursor*
- ~~corner~~ *— attribute of Rectangle*
- ~~time~~ *— an implicit attribute of the system*

Candidate Classes

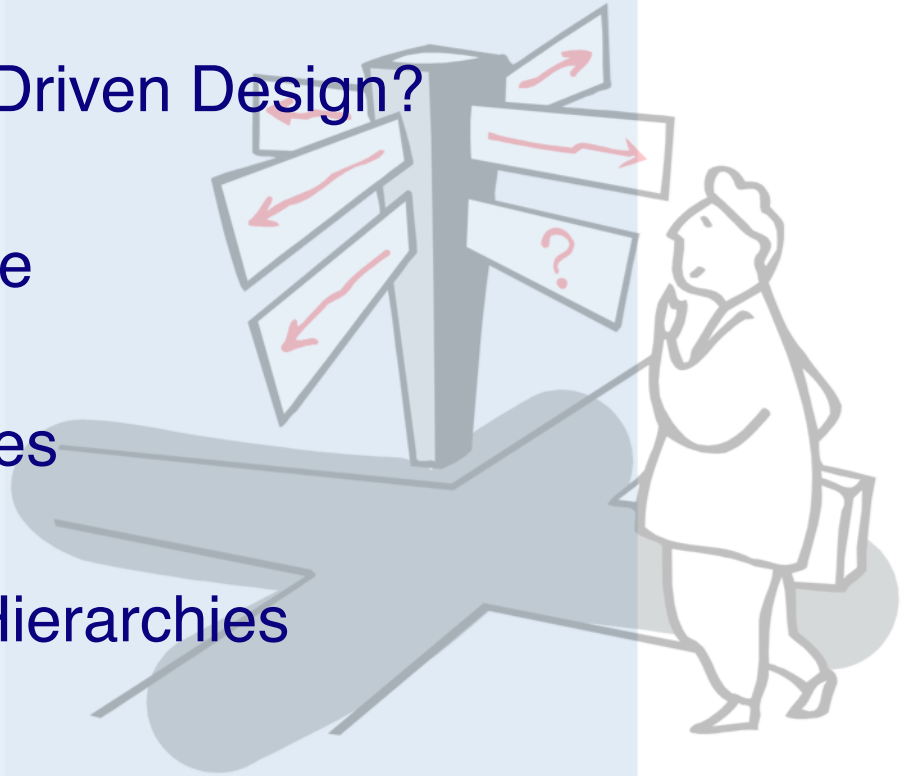
Preliminary analysis yields the following candidates:

Character	Line Element
Control Point	Point
Creation Tool	Rectangle
Current Selection	Rectangle Creation Tool
Drawing	Rectangle Element
Drawing Editor	Selection Tool
Drawing Element	Text Creation Tool
Ellipse Creation Tool	Text Element
Ellipse Element	Tool
Line Creation Tool	

Expect the list to evolve as design progresses.

Roadmap

- Why use Responsibility-Driven Design?
- Finding Classes
- Class Selection Rationale
- **CRC sessions**
- Identifying Responsibilities
- Finding Collaborations
- Structuring Inheritance Hierarchies



CRC Cards

Use CRC cards to record candidate classes:

Text Creation Tool	*subclass of Tool*
Editing Text	Text Element

Record the candidate *Class Name* and *superclass* (if known)

Record each *Responsibility* and the *Collaborating classes*

— compact, easy to manipulate, easy to modify or discard!

— easy to arrange, reorganize

— easy to retrieve discarded classes

CRC Sessions

CRC cards are *not* a specification of a design.

They are a tool to *explore* possible designs.

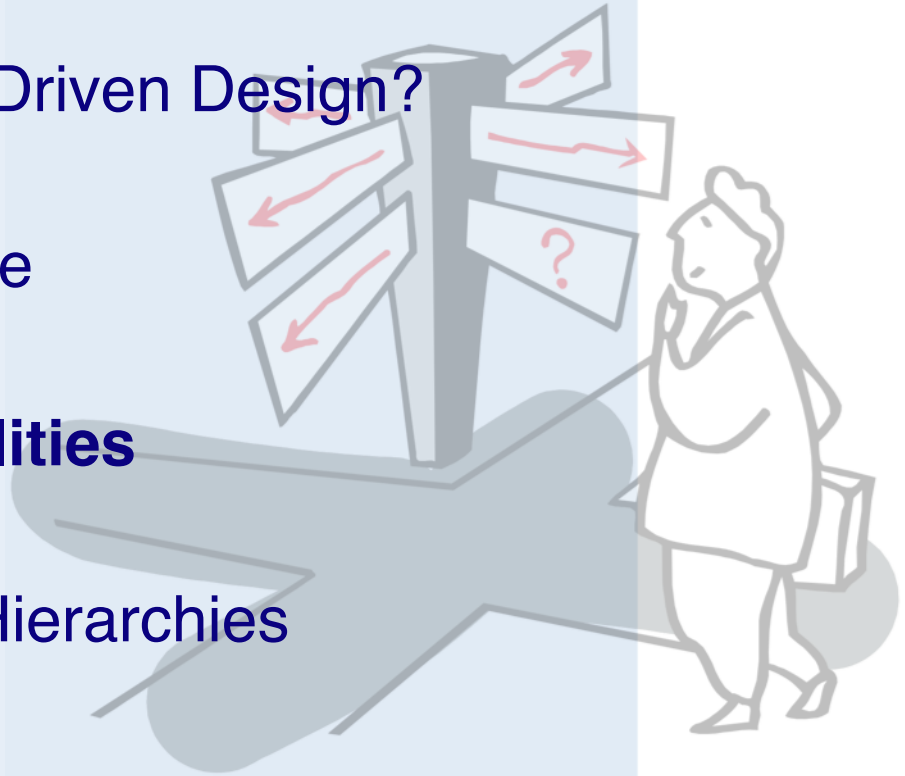
- Prepare a CRC card for *each candidate class*
- Get a team of Developers to *sit around a table* and distribute the cards to the team
- The team *walks through scenarios*, playing the roles of the classes.

This exercise will uncover:

- *unneeded* classes and responsibilities
- *missing* classes and responsibilities

Roadmap

- Why use Responsibility-Driven Design?
- Finding Classes
- Class Selection Rationale
- CRC sessions
- **Identifying Responsibilities**
- Finding Collaborations
- Structuring Inheritance Hierarchies



Responsibilities

What are responsibilities?

- —the knowledge an object maintains and provides
- —the actions it can perform

Responsibilities represent the *public services* an object may provide to clients (but not the way in which those services may be implemented)

- —specify *what* an object does, not *how* it does it
- —don't describe the interface yet, only *conceptual responsibilities*

Identifying Responsibilities

- Study the requirements specification:
  - highlight *verbs* and determine which represent responsibilities
  - perform a *walk-though* of the system
    - *explore as many scenarios as possible*
    - *identify actions resulting from input to the system*

- Study the candidate classes:
  - class names ⇒ roles ⇒ responsibilities
  - recorded purposes on class cards ⇒ responsibilities

How to assign responsibility?

Pelrine's Laws:

> "Don't do anything you can push off to someone else."

> "Don't let anyone else play with you."

Assigning Responsibilities

- *Evenly distribute* system intelligence
  - avoid procedural centralization of responsibilities
  - keep responsibilities close to objects rather than their clients
- State responsibilities as *generally* as possible
  - “draw yourself” vs. “draw a line/rectangle etc.”
  - leads to sharing
- Keep *behaviour* together with any *related information*
  - principle of encapsulation

Assigning Responsibilities ...

- Keep information about one thing in *one place*
  - if multiple objects need access to the same information
    1. *a new object may be introduced to manage the information, or*
    2. *one object may be an obvious candidate, or*
    3. *the multiple objects may need to be collapsed into a single one*
- *Share* responsibilities among related objects
  - break down complex responsibilities

Relationships Between Classes

Additional responsibilities can be uncovered by examining relationships between classes, especially:

- The “Is-Kind-Of” Relationship:
  - classes sharing a *common attribute* often share a *common superclass*
  - common superclasses suggest *common responsibilities*

e.g., to create a new Drawing Element, a Creation Tool must:

1. accept user input — *implemented in subclass*
2. determine location to place it — *generic*
3. instantiate the element — *implemented in subclass*

Relationships Between Classes ...

- The “Is-Analogous-To” Relationship:
  - *similarities* between classes suggest as-yet-undiscovered superclasses
- The “Is-Part-Of” Relationship:
  - *distinguish* (don’t share) responsibilities of *part* and of *whole*

Difficulties in assigning responsibilities suggest:

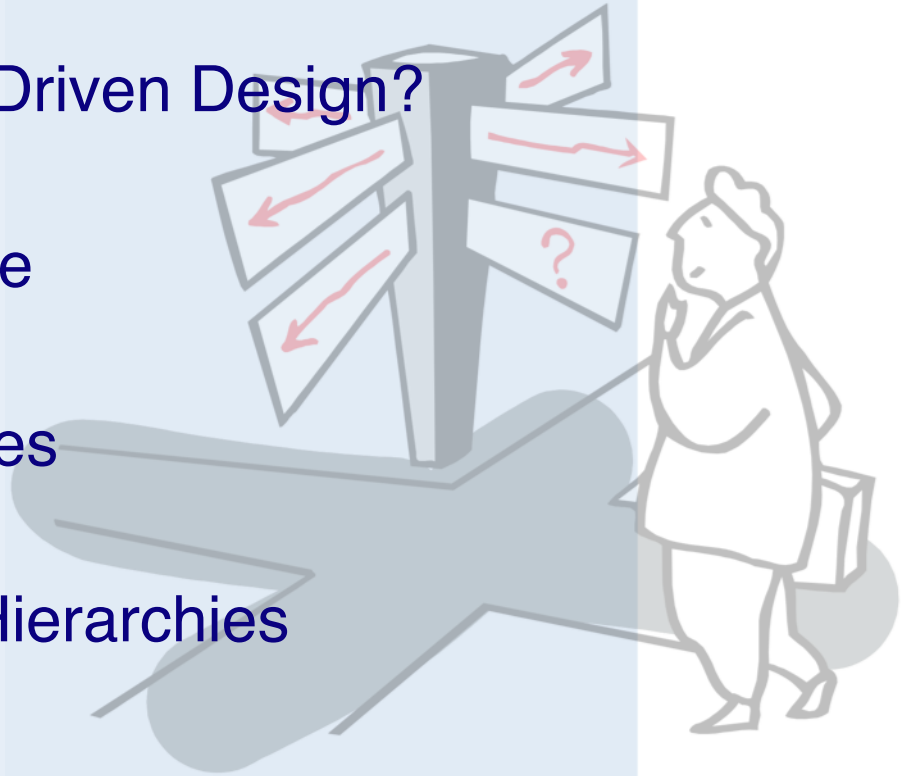
- *missing classes* in design, or — e.g., Group Element
- *free choice* between multiple classes

Example Relationships

- Drawing element *is-part-of* Drawing
- Drawing Element *has-knowledge-of* Control Points
- Rectangle Tool *is-kind-of* Creation Tool

Roadmap

- Why use Responsibility-Driven Design?
- Finding Classes
- Class Selection Rationale
- CRC sessions
- Identifying Responsibilities
- **Finding Collaborations**
- Structuring Inheritance Hierarchies



Collaborations

What are collaborations?

- *collaborations* are *client requests* to servers needed to fulfil responsibilities
- collaborations reveal *control and information flow* and, ultimately, subsystems
- collaborations can uncover *missing responsibilities*
- analysis of communication patterns can reveal *misassigned* responsibilities

Finding Collaborations

For each responsibility:

1. Can the class *fulfil* the responsibility *by itself?*
2. If not, *what does it need*, and from what other class can it obtain what it needs?

For each class:

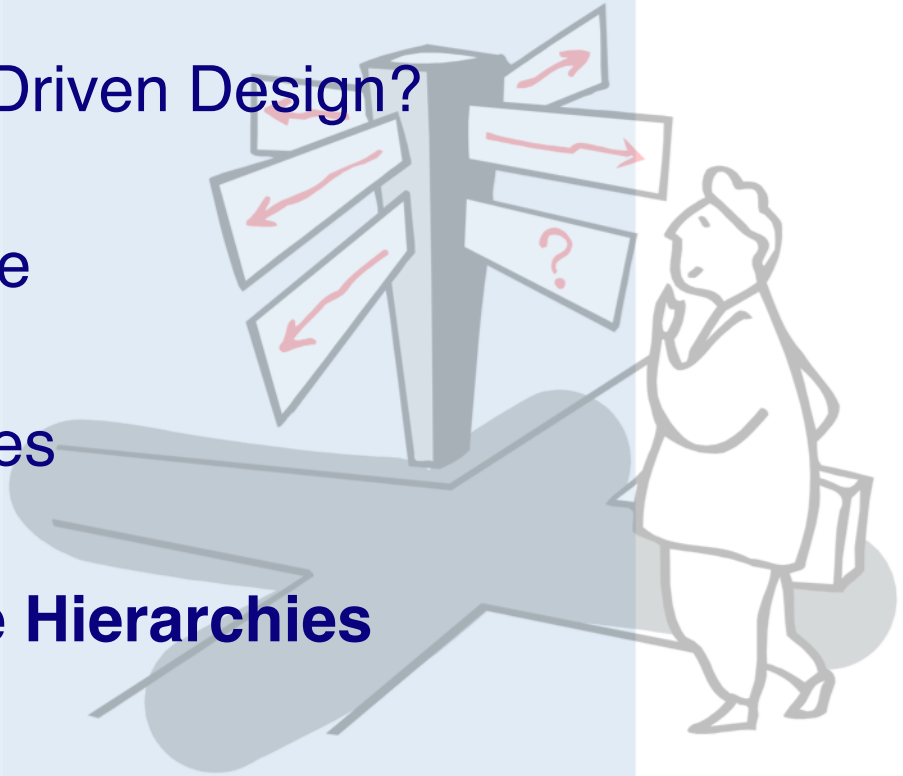
1. What does this class *know*?
2. What *other classes* need its information or results? Check for collaborations.
3. Classes that *do not interact* with others should be *discarded*. (Check carefully!)

Listing Collaborations

Drawing	
Knows which elements it contains	
Maintains order of elements	Drawing Element

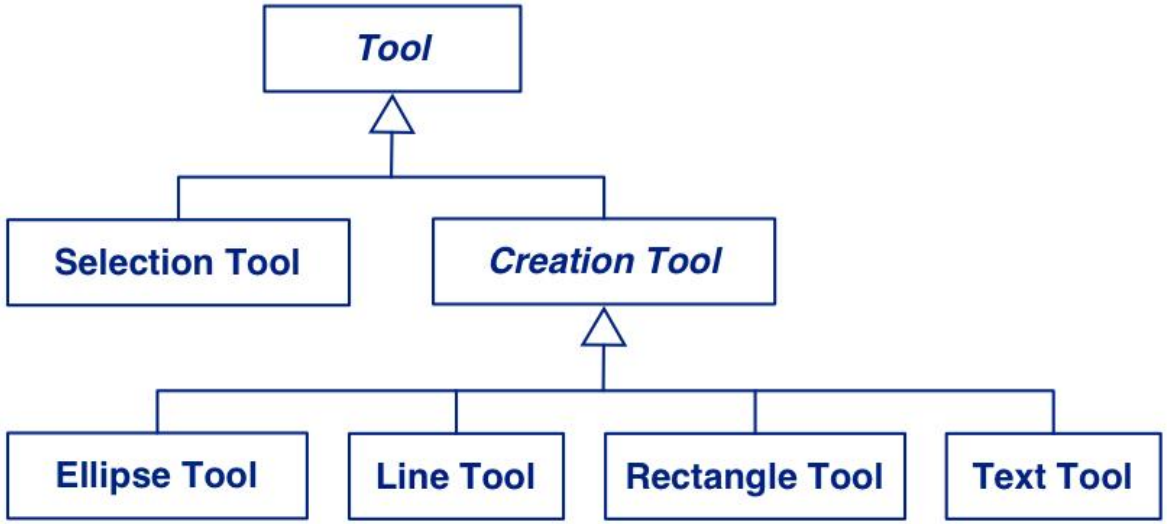
Roadmap

- Why use Responsibility-Driven Design?
- Finding Classes
- Class Selection Rationale
- CRC sessions
- Identifying Responsibilities
- Finding Collaborations
- **Structuring Inheritance Hierarchies**



Finding Abstract Classes

Abstract classes factor out common behaviour shared by other classes



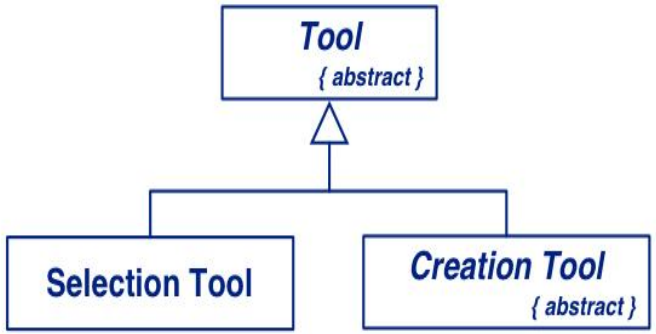
- group related classes with common attributes
- introduce abstract superclasses to represent the group
- “categories” are good candidates for abstract classes

Warning: beware of premature classification; your hierarchy will evolve!

Sharing Responsibilities

Concrete classes may be both instantiated and inherited from.

Abstract classes may only be inherited from.



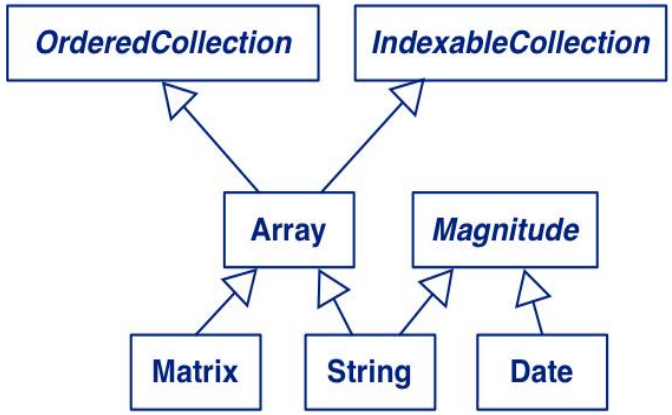
Note on class cards and on class diagram.

Venn Diagrams can be used to visualize shared responsibilities.

(Warning: not part of UML!)

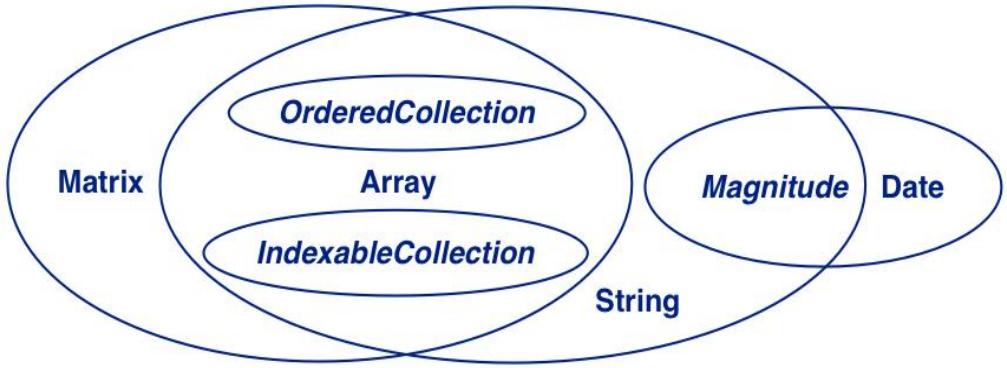
Multiple Inheritance

Decide whether a class will be *instantiated* to determine if it is *abstract or concrete*.



Responsibilities of subclasses are *larger* than those of *superclasses*.

Intersections represent *common superclasses.*



Building Good Hierarchies

Model a "kind-of" hierarchy:

- Subclasses should *support all inherited responsibilities*, and possibly more

Factor common responsibilities as high as possible:

- Classes that *share common responsibilities* should *inherit from a common abstract superclass*; introduce any that are missing

Building Good Hierarchies …

Make sure that abstract classes do not inherit from concrete classes:

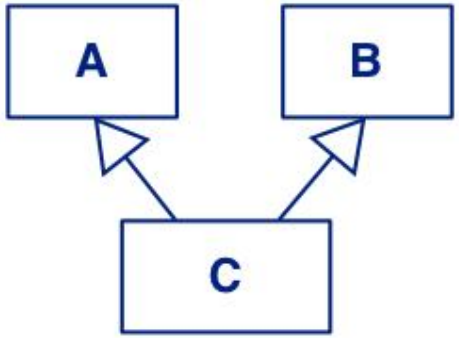
- Eliminate by introducing *common abstract superclass*: abstract classes should support responsibilities in an implementation-independent way

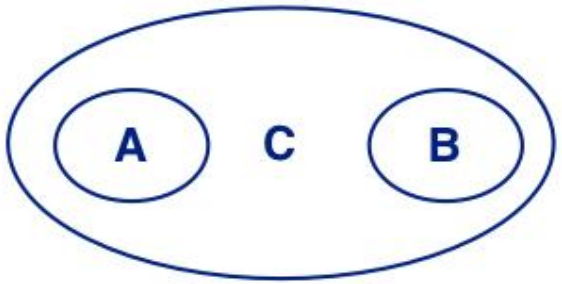
Eliminate classes that do not add functionality:

- Classes should either add new responsibilities, or a particular way of implementing inherited ones

Building Kind-Of Hierarchies

Correctly Formed Subclass Responsibilities:



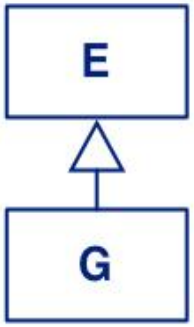


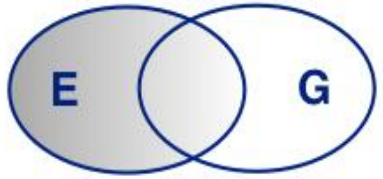
C assumes *all* the responsibilities of both A and B

Building Kind-Of Hierarchies ...

Incorrect Subclass/Superclass Relationships

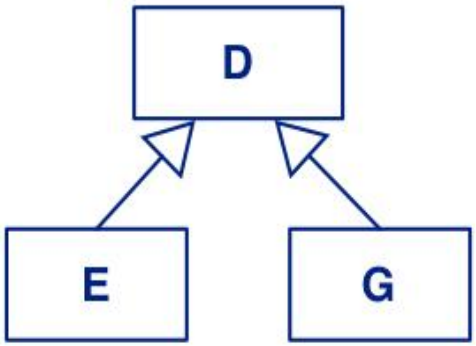
> G assumes only *some* of the responsibilities inherited from E

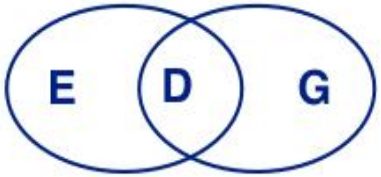




Revised Inheritance Relationships

> Introduce *abstract superclasses* to encapsulate common responsibilities

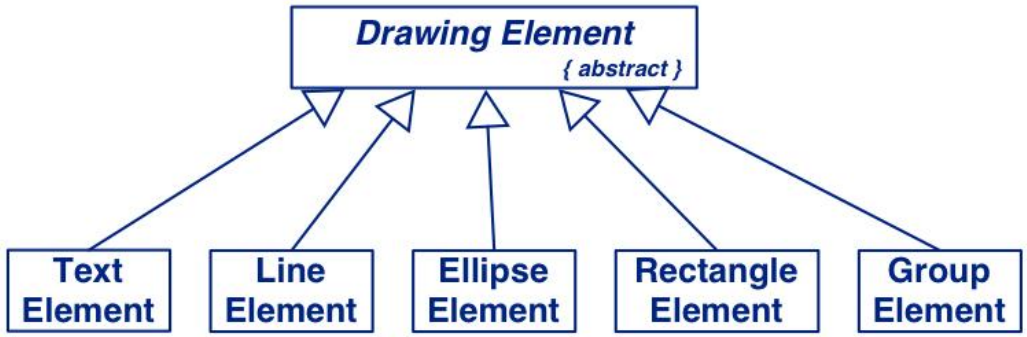


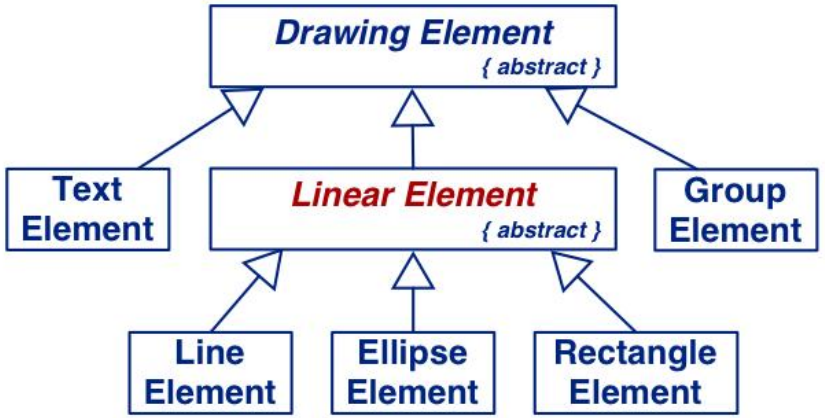


Refactoring Responsibilities

Lines, Ellipses and Rectangles are responsible for keeping track of the width and colour of the lines they are drawn with.

This suggests a *common superclass.*





Protocols

A protocol is a *set of signatures* (i.e., an *interface*) to which a class will respond.

- Generally, protocols are specified for *public responsibilities*
- Protocols for *private* responsibilities should be specified if they will be used or implemented by *subclasses*

1. Construct protocols for each class
2. Write a design specification for each class and subsystem
3. Write a design specification for each contract

What you should know!

- What criteria can you use to identify potential classes?
- How can CRC cards help during analysis and design?
- How can you identify abstract classes?
- What are class responsibilities, and how can you identify them?
- How can identification of responsibilities help in identifying classes?
- What are collaborations, and how do they relate to responsibilities?
- How can you identify abstract classes?
- What criteria can you use to design a good class hierarchy?
- How can refactoring responsibilities help to improve a class hierarchy?

Can you answer the following questions?

- When should an attribute be promoted to a class?
- Why is it useful to organize classes into a hierarchy?
- How can you tell if you have captured all the responsibilities and collaborations?
- What use is multiple inheritance during design if your programming language does not support it?

License