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
> SOLID object-oriented design principles
Roadmap

> Why use Responsibility-Driven Design?
> Finding Classes
> Class Selection Rationale
> CRC sessions
> Identifying Responsibilities
> Finding Collaborations
> Structuring Inheritance Hierarchies
> SOLID object-oriented design principles
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
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?
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
- SOLID object-oriented design principles
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
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 modeling of inheritance
- Model *interfaces* to the system — e.g., user interface, program interfaces
- Model attribute *values*, not attributes — e.g., Point vs. Centre
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.
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
> SOLID object-oriented design principles
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
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
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
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
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:*

<table>
<thead>
<tr>
<th>Class</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>Line Element</td>
</tr>
<tr>
<td>Control Point</td>
<td>Point</td>
</tr>
<tr>
<td>Creation Tool</td>
<td>Rectangle</td>
</tr>
<tr>
<td>Current Selection</td>
<td>Rectangle Creation Tool</td>
</tr>
<tr>
<td>Drawing</td>
<td>Rectangle Element</td>
</tr>
<tr>
<td>Drawing Editor</td>
<td>Selection Tool</td>
</tr>
<tr>
<td>Drawing Element</td>
<td>Text Creation Tool</td>
</tr>
<tr>
<td>Ellipse Creation Tool</td>
<td>Text Element</td>
</tr>
<tr>
<td>Ellipse Element</td>
<td>Tool</td>
</tr>
<tr>
<td>Line Creation Tool</td>
<td></td>
</tr>
</tbody>
</table>

*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
- SOLID object-oriented design principles
Use CRC cards to record candidate classes:

<table>
<thead>
<tr>
<th>Text Creation Tool</th>
<th>subclass of Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Editing Text</td>
<td>Text Element</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
- SOLID object-oriented design principles
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-through* 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

each class exists because of an implied responsibility
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

A Drawing Editor knows when the drawing has changed; the Drawing knows which elements to display; each Drawing Element knows how and where its presentation should be drawn...
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*
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
> SOLID object-oriented design principles
What are collaborations?

> **collaborations** are *client requests* to servers needed to fulfill 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 fulfill 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!)
<table>
<thead>
<tr>
<th>Drawing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Knows which elements it contains</td>
</tr>
<tr>
<td>Maintains order of elements</td>
</tr>
</tbody>
</table>
Roadmap

- Why use Responsibility-Driven Design?
- Finding Classes
- Class Selection Rationale
- CRC sessions
- Identifying Responsibilities
- Finding Collaborations
- **Structuring Inheritance Hierarchies**
- SOLID object-oriented design principles
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!
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
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
C assumes *all* the responsibilities of both A and B.
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.
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
Roadmap

> Why use Responsibility-Driven Design?
> Finding Classes
> Class Selection Rationale
> CRC sessions
> Identifying Responsibilities
> Finding Collaborations
> Structuring Inheritance Hierarchies
> SOLID object-oriented design principles
SOLID (object-oriented design principles)

> **Single responsibility**
> **Open-closed**
> **Liskov substitution**
> **Interface segregation**
> **Dependency inversion**

*Concerns*: Rigidity, fragility, immobility, viscosity (!)

http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

rigidity: hard to change; fragility: breaks when changed; immobility: impossible to reuse (too many dependencies); viscosity: changes break design
Every class should have a *single responsibility*

*There should never be more than one reason for a class to change*

If a class has multiple responsibilities, then they become coupled. A change to one responsibility will then impact another. Difficulty: what is the granularity of a “responsibility”? NB: Martin equates this principle with “cohesion”
Open/closed principle

Software entities should be *open for extension*, but *closed for modification*.

“In other words, we want to be able to change what the modules do, without changing the source code of the modules.”

Example: Class — instantiate as an encapsulated object; extend as a subclass
Component — fixed interface; hooks to plug in new behaviour (cf eclipse)
Liskov substitution principle

(Instances of) subclasses should be *substitutable* for (instances of) their base classes.

Restated in terms of *contracts*, a derived class is substitutable for its base class if:

- *Its preconditions are no stronger than the base class method.*
- *Its postconditions are no weaker than the base class method.*


Example: Is a Circle an Ellipse? Depends on the contract clients expect! (Ditto for Square and Rectangle.) Wegner and Zdonik actually introduced the “Principle of Substitutability” several years earlier. Note that Liskov and Wing actually refer to a much stronger notion of behavioral substitutability than Uncle Bob (or Wegner and Zdonik do), and is much stronger than what OO programs usually require. It all depends on how strong your type system is!
Interface segregation principle

Many *client-specific interfaces* are better than one general purpose interface.

*Clients should not be forced to depend upon interfaces that they don't use.*

The point is to reduce coupling, and thus to avoid rippling changes to many clients when interfaces inevitably change. The problem can be solved either by allowing classes to implement multiple interfaces (as in Java), or by delegation (interposing an Adapter).

Dependency inversion principle

Depend upon abstractions.
Do not depend upon concretions.

*High-level modules should not depend on low-level modules. Both should depend on abstractions.*

*Abstractions should not depend upon details. Details should depend upon abstractions.*


Decouple high-level code from low-level code so details can change!
Can be solved by subclassing, plugins, code generation, dependency injection ...
Problem: you need a concrete class when you create instances.
Solution: use an Abstract Factory!
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?
Attribution-ShareAlike 3.0

You are free:

▪ to copy, distribute, display, and perform the work
▪ to make derivative works
▪ to make commercial use of the work

Under the following conditions:

BY: Attribution. You must attribute the work in the manner specified by the author or licensor.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

▪ For any reuse or distribution, you must make clear to others the license terms of this work.
▪ Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

http://creativecommons.org/licenses/by-sa/3.0/