b
UNIVERSITÄT
BERN

Introduction to Software Engineering

4. GOOD: Good Object Oriented Design

Mircea F. Lungu

Based on a lecture by Oscar Nierstrasz.

Bibliography

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

- **The Early History of Smalltalk**, A. Kay, *In History of Programming Languages, 1993*

- **Design Principles and Design Patterns**, *R.C. Martin, 2000*

- **Object-Oriented Design Heuristics**, *A.Riel, 2000*

- **Ten things I hate about OOP,** *Oscar Nierstrasz, JOT, 2010*

- **The Power of Interoperability: Why Objects Are Inevitable**, *J. Aldrich, Onward! 2013*

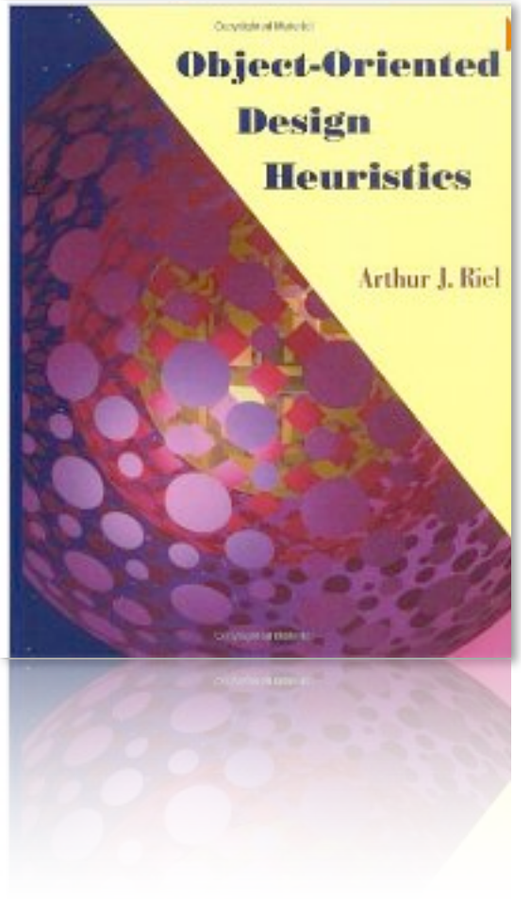
Roadmap

- Responsibility-Driven Design
  - **Finding Classes**
  - [CRC sessions]
  - Identifying Responsibilities
  - Finding Collaborations
  - Structuring Inheritance Hierarchies
- Object-oriented design principles

Object-Oriented Decomposition

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

Design is not algorithmic



60 design heuristics

This is a Summary of the heuristics in the OO Design Heuristics by Arthur Riel.

(A) Classes and Objects

1. All data should be hidden within its class
2. Users of a class must be dependent on its public interface, but a class should not be dependent on its users.
3. Minimize the number of messages in the protocol of a class (protocol of a class means the set of messages to which an instance of the class can respond)
4. Implement a minimal public interface that all classes understand
5. Do not put implementation details such as common-code private functions into the public interface of a class
6. Do not clutter the public interface of a class with things that users of that class are not able to use or are not interested in using.
7. Classes should only exhibit nil or export coupling with other classes, that is, a class should only use operations in the public interface of another class or have nothing to do with that class.
8. A class should capture one and only one key abstraction
9. Keep related data and behaviour in one place.
10. Spin off nonrelated information into another class (that is, non-communicating behaviour)
11. Be sure the abstractions that you model are classes and not simply the roles objects play

(B) Topologies of Action-Oriented Versus Object-Oriented Applications

12. Distribute system intelligence horizontally as uniformly as possible, that is the top level classes in a design should share the work uniformly.

13. Do not create god classes or god objects in your system. Be very suspicious of a class whose name contains DRIVER, MANGER, SYSTEM, SUBSYSTEM, etc.

14. Beware of classes that have many accessor methods defined in their interface. Having many implies that related data and behaviour are not being kept in one place.

15. Beware of classes that have too much non-communicating behaviour, that is, methods that operate on a proper subset of the data members of a class. God classes often exhibit a great deal of non-communicating behaviour.

16. In applications that consist of an object oriented model interacting with a user interface, the model should never be dependent on the interface. The interface should be dependent on the model.

17. Model the real world whenever possible. (This heuristic is often violated for reasons of system intelligence distribution, avoidance of god classes, and the keeping of related data and behaviour in one place.)

18. Eliminate irrelevant classes from your design.

19. Eliminate classes that are outside the system.

20. Do not turn an operation into a class. Be suspicious of any class whose name is a verb or is derived from a verb, especially those which have only one piece of meaningful behaviour. Ask if that piece of meaningful behaviour needs to be migrated to some existing or undiscovered class.

21. Agent classes are often placed in the analysis model of an application. During design time, many agents are found to be irrelevant and should be removed.

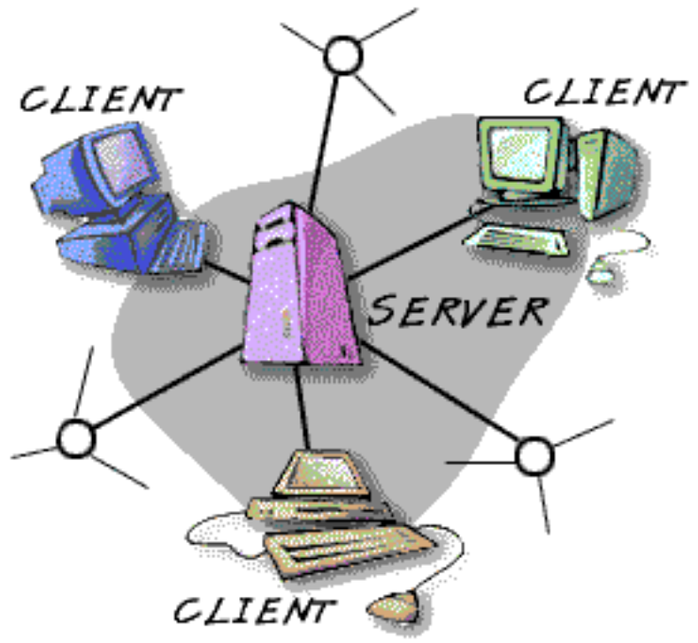
Expertise matters

A good *sense of style* often helps produce clean, elegant designs

Alan Kay on OO

What I got from Simula was that you could now replace bindings and assignment with *goals*. [...] the objects should be presented as ***sites of higher level behaviors more appropriate for use as dynamic components***.

Responsibility-Driven Design



Every “**object would be a server offering services**” that are accessed via messages to the object.

A. Kay

RDD Steps

1. Find the *classes* in your system

2. Determine the *responsibilities* of each class

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

4. *Factor* common responsibilities to build class hierarchies

Roadmap

- Responsibility-Driven Design
  - **Finding Classes**
  - [CRC sessions]
  - Identifying Responsibilities
  - Finding Collaborations
  - Structuring Inheritance Hierarchies
- Object-oriented design principles

Finding Classes

Start with requirements specification:

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

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.

Let’s find classes (a collabortive slide)

- IDrawingElement: Line, Ellipsis, Text, Bounding Box,
- Selection
- ATools: SelectionTool, CreationTool
- A/I-CreationTool: LineCreationTool, ...
- Point

Finding Classes ...

1. Look for *noun phrases*
   - separate into obvious classes, uncertain candidates, and nonsense
2. Refine to a list of *candidate classes*

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.

Class Selection Rationale

Look for 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

- Responsibility-Driven Design
  - Finding Classes
  - **[CRC sessions]**
  - Identifying Responsibilities
  - Finding Collaborations
  - Structuring Inheritance Hierarchies
- SOLID object-oriented design principles

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 *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

- Responsibility-Driven Design
  - Finding Classes
  - [CRC sessions]
  - **Identifying Responsibilities**
  - Finding Collaborations
  - Structuring Inheritance Hierarchies
- SOLID object-oriented design principles

What are object responsibilities?

—the **knowledge** an object maintains and provides

—the **actions** it can perform

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

How to assign responsibilities?

Assigning Responsibilities: Be lazy

“Don't do anything you can push off to someone else.”

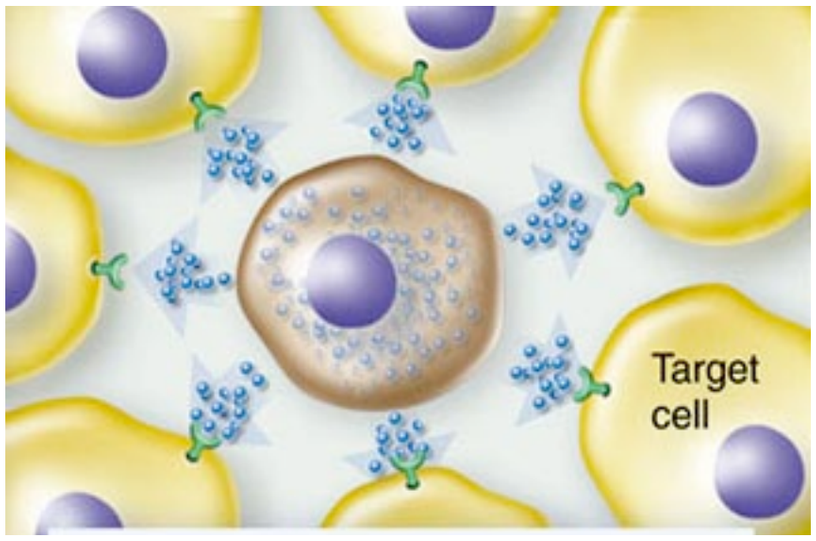
(Pelrine)

Assigning Responsibilities: Be tough

“Don't let anyone else play with your toys”.

(Pelrine)

Assigning Responsibilities: Be socialist



Evenly distribute **system intelligence**

Assigning Responsibilities

State responsibilities as *generally* as possible

Assigning Responsibilities

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*

Roadmap

- Responsibility-Driven Design
  - Finding Classes
  - [CRC sessions]
  - Identifying Responsibilities
  - **Finding Collaborations**
  - Structuring Inheritance Hierarchies
- SOLID object-oriented design principles

Relationships Between Classes

- Drawing element ***is-part-of*** Drawing

- Drawing Element ***has-knowledge-of*** Control Points

- Rectangle Tool ***is-kind-of*** Creation Tool

Relationships Between Classes

- 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-Part-Of” Relationship:
  - —*distinguish* (don’t share) responsibilities of *part* and of *whole*

Relationships Between Classes ...

- The "Is-Analogous-To" Relationship:
  - —*similarities* between classes suggest as-yet-undiscovered superclasses

Difficulties in assigning responsibilities suggest:

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

Collaborations

What are collaborations?

- *collaborations* are *client requests* to servers needed to fulfill responsibilities
- collaborations reveal *control and information flow* and, ultimately, subsystems

- 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!)

Listing Collaborations

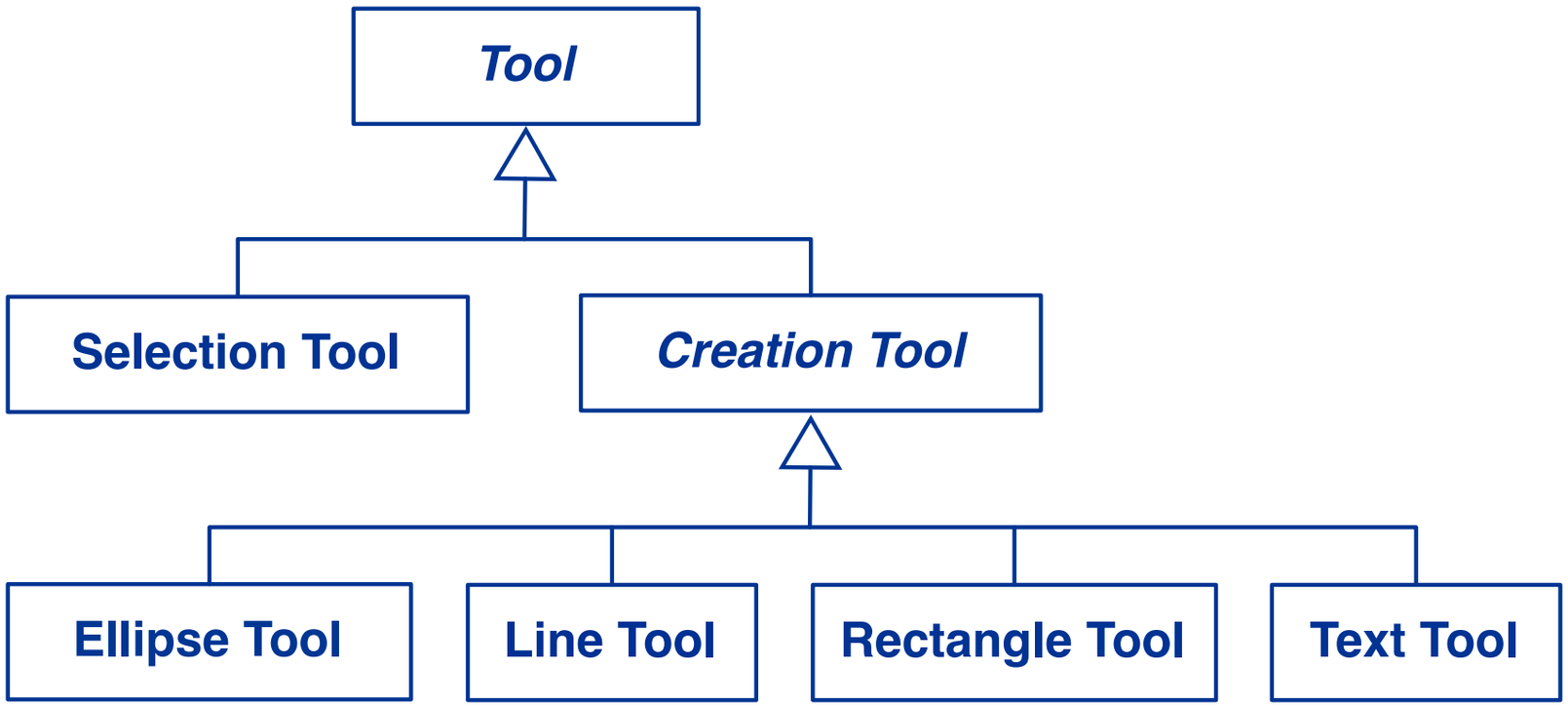
Drawing	
Knows which elements it contains	
Maintains order of elements	Drawing Element

Roadmap

- Responsibility-Driven Design
  - Finding Classes
  - [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



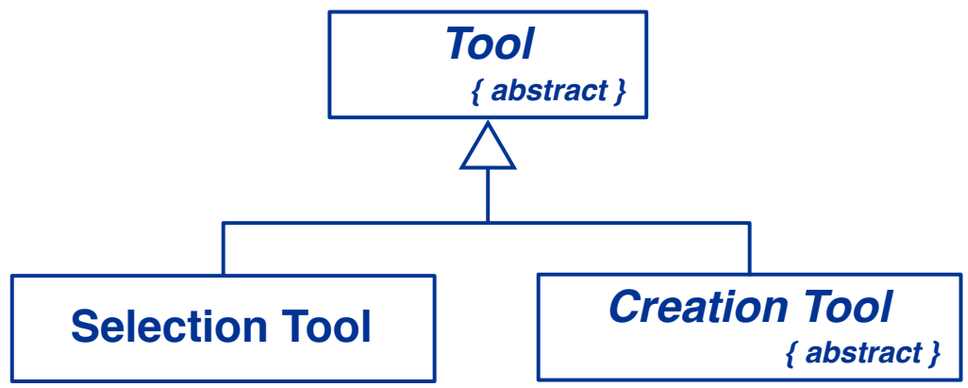
- group related classes with common attributes
- introduce abstract superclasses to represent the group
- “categories” are good candidates for abstract classes

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

Sharing Responsibilities

Concrete classes may be both instantiated and inherited from.

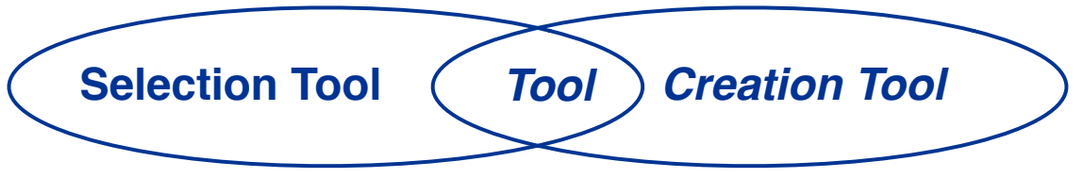
Abstract classes may only be inherited from.



Note on class cards and on class diagram.

Venn Diagrams can be used to visualize shared responsibilities.

(Warning: not part of UML!)



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 …

Abstract classes do not inherit from concrete classes:

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

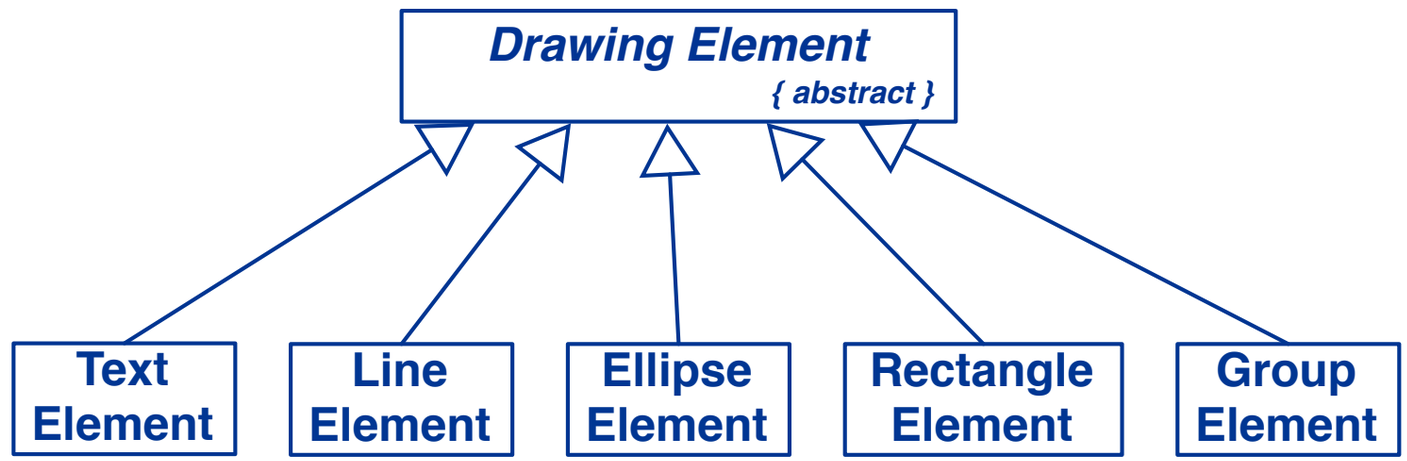
Eliminate classes that do not add functionality:

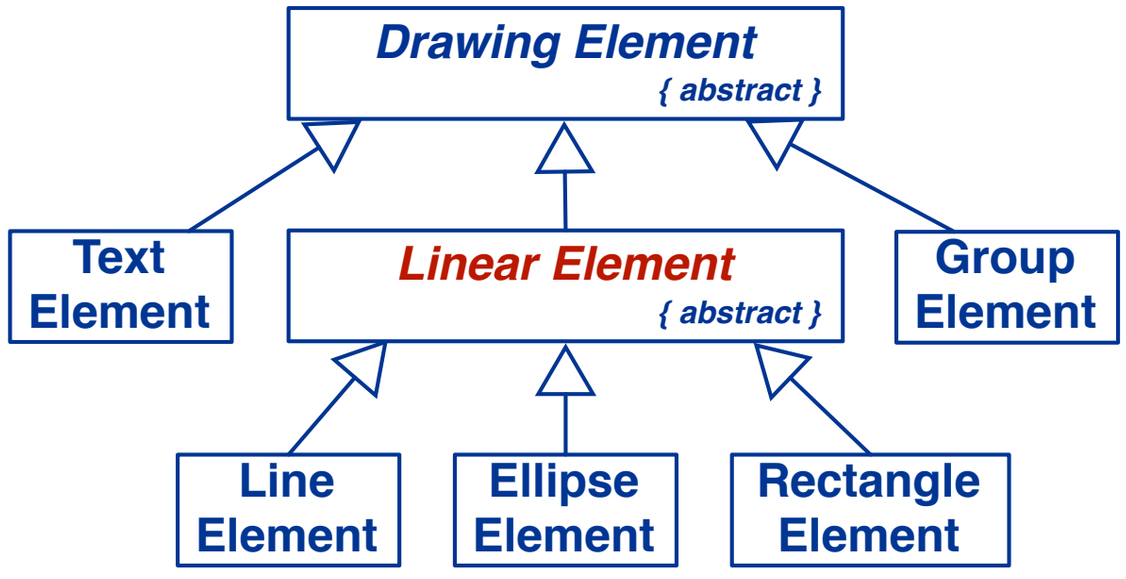
- Classes should either add new responsibilities, or a particular way of implementing inherited ones

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.*





Roadmap

- Responsibility-Driven Design
  - Finding Classes
  - [CRC sessions]
  - Identifying Responsibilities
  - Finding Collaborations
  - Structuring Inheritance Hierarchies
- **SOLID object-oriented design principles**

SOLID (object-oriented design principles)

- **S**ingle responsibility
- **O**pen-closed
- **L**iskov substitution
- **I**nterface segregation
- **D**ependency inversion

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

Robert C. Martin. *Design Principles and Design Patterns*. 2000.
http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

Single responsibility principle (Read!)

Every class should have a *single responsibility*

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

Robert C. Martin. *SRP: The Single Responsibility Principle*. 2000. http://www.objectmentor.com/resources/articles/srp.pdf (!)

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."

Bertrand Meyer, *Object-Oriented Software Construction*, 1988.
See also: http://www.objectmentor.com/resources/articles/ocp.pdf

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.*

Barbara Liskov, Jeannette M. Wing. *A behavioral notion of subtyping*. ACM TOPLAS, 1994.
http://www.cse.ohio-state.edu/~neelam/courses/788/lwb.pdf

Peter Wegner, Stanley Zdonik. *Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like*. ECOOP 1988.
http://www.ifs.uni-linz.ac.at/~ecoop/cd/tocs/t0322.htm

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.

Robert C. Martin, *The Interface Segregation Principle*, C++ Report, June 1996.
http://www.objectmentor.com/resources/articles/isp.pdf

Dependency inversion principle (Read!!)

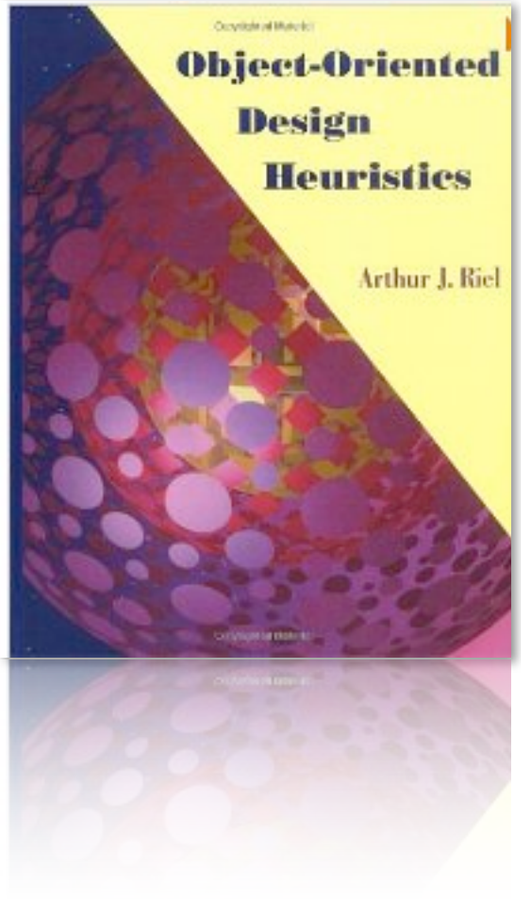
Depend upon abstractions.
Do not depend upon concretions.

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.

Robert C. Martin, *The Dependency Inversion Principle*. C++ Report, May 1996.
http://www.objectmentor.com/resources/articles/dip.pdf

To read!



60 design heuristics

This is a Summary of the heuristics in the OO Design Heuristics by Arthur Riel.

(A) Classes and Objects

1. All data should be hidden within its class
2. Users of a class must be dependent on its public interface, but a class should not be dependent on its users.
3. Minimize the number of messages in the protocol of a class (protocol of a class means the set of messages to which an instance of the class can respond)
4. Implement a minimal public interface that all classes understand
5. Do not put implementation details such as common-code private functions into the public interface of a class
6. Do not clutter the public interface of a class with things that users of that class are not able to use or are not interested in using.
7. Classes should only exhibit nil or export coupling with other classes, that is, a class should only use operations in the public interface of another class or have nothing to do with that class.
8. A class should capture one and only one key abstraction
9. Keep related data and behaviour in one place.
10. Spin off nonrelated information into another class (that is, non-communicating behaviour)
11. Be sure the abstractions that you model are classes and not simply the roles objects play

(B) Topologies of Action-Oriented Versus Object-Oriented Applications

12. Distribute system intelligence horizontally as uniformly as possible, that is the top level classes in a design should share the work uniformly.

13. Do not create god classes or god objects in your system. Be very suspicious of a class whose name contains DRIVER, MANGER, SYSTEM, SUBSYSTEM, etc.

14. Beware of classes that have many accessor methods defined in their interface. Having many implies that related data and behaviour are not being kept in one place.

15. Beware of classes that have too much non-communicating behaviour, that is, methods that operate on a proper subset of the data members of a class. God classes often exhibit a great deal of non-communicating behaviour.

16. In applications that consist of an object oriented model interacting with a user interface, the model should never be dependent on the interface. The interface should be dependent on the model.

17. Model the real world whenever possible. (This heuristic is often violated for reasons of system intelligence distribution, avoidance of god classes, and the keeping of related data and behaviour in one place.)

18. Eliminate irrelevant classes from your design.

19. Eliminate classes that are outside the system.

20. Do not turn an operation into a class. Be suspicious of any class whose name is a verb or is derived from a verb, especially those which have only one piece of meaningful behaviour. Ask if that piece of meaningful behaviour needs to be migrated to some existing or undiscovered class.

21. Agent classes are often placed in the analysis model of an application. During design time, many agents are found to be irrelevant and should be removed.

Design is iterative

There is no great writing, only great rewriting.

L. Brandeis

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?

creative commons

COMMONS DEED

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:

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/