SWINBURNE UNIVERSITY OF TECHNOLOGY

SWE30003
Software Architectures and Design

Lecture 6
Object Design (Part II) –
Design Refinement / Better Design

1

Logistical matters

- Weekly submissions – A & Q
  - Week 2: 214 and 202 out of 270;
  - Week 3: 222 and 215 out of 270;
  - Week 4: 202 and 199 out of 266;
  - Week 5:
  - Week 6:
  - Still have x student who did not submit anything!!
  - Note that this is a hurdle requirement; no late submission
- Assignment 1: due end of this week.
- Assignment 2 spec: To be released by end of the week; start as soon as released (inc semester break).

2

Question to Answer from week 5

In what way are objects, the artificial building blocks of a machine or software system, similar to a living, biological organism? What role do they play in helping us to understand and think about a problem?

3

3

Principal References

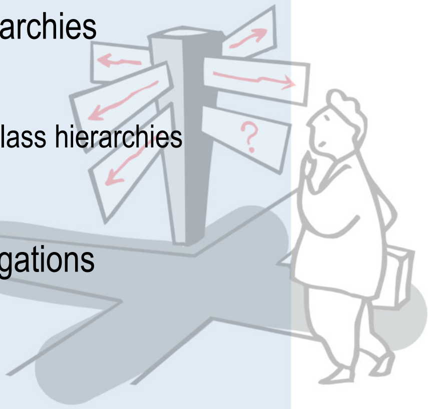
- Bernd Bruegge and Allen H. Dutoit, *Object-oriented Software Engineering*, Prentice Hall, 2001, Chapter 5.
- Bertrand Meyer, *Object-Oriented Software Construction* (2nd Edition), Prentice Hall, 1997, Chapters 11, 23, 24.
- Rebecca Wirfs-Brock, Brian Wilkerson and Lauren Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990, Chapter 5.
- Arthur J. Riel, *Object-Oriented Design Heuristics*, Addison Wesley, 1996, various chapters.

4

4

Roadmap

- **Recap Lecture 5**
- Building good class hierarchies
  - Types of inheritance
  - Heuristics for creating class hierarchies
- Roles and Protocols
- Associations and Aggregations
- Design by Contract



5

5

Recap from Lecture 5

An application = a set of interacting *objects*

An object = an implementation of one or more *roles*

A role = a set of related *responsibilities*

A responsibility = an obligation to perform a task or know certain information

A collaboration = an *interaction* of objects *via* roles

6

6

Recap from Lecture 5 (cont.)

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.
- ☐ Do not describe the interface yet, only *conceptual responsibilities.*

7

7

Recap from Lecture 5 (cont.)

What are collaborations?

- *collaborations* are *client requests* to servers needed to fulfill responsibilities,
  (here: both client & server are objects/classes)
- collaborations reveal *control and information flow*
- collaborations can uncover *missing responsibilities*,
- analysis of communication patterns can reveal *mis-assigned* responsibilities.

8

8

Recap from Lecture 5 (cont.)

- Try to *evenly distribute* system intelligence
  - avoid procedural centralization of responsibilities
  - keep responsibilities close to objects rather than their clients
  - ☞ *Note: this is not always possible (or even desirable)!*
- State responsibilities as *generally* as possible
  - "draw yourself" vs. "draw a line/rectangle etc."
  - ☞ leads to sharing
- Keep *behavior* together with any *related information*
  - principle of *encapsulation*.

9

9

What we got…

- A list of candidate classes
  - and some initial justification why they are needed
- A list of responsibilities
  - and an initial allocation of responsibilities to classes
- For each responsibility assigned to a class
  - a list of collaborators needed to fulfill this responsibility

☞ Expanded domain model with design solution (more classes), extended with responsibilities and collaborations…

10

10

What next? … Refine the design!

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?
   - ☐ Can we identify protocols and roles amongst the collaborations?
   - ☐ Can we identify *contracts* between collaboration objects?
3. Apply *design heuristics* and *common sense* to improve specific design aspects.

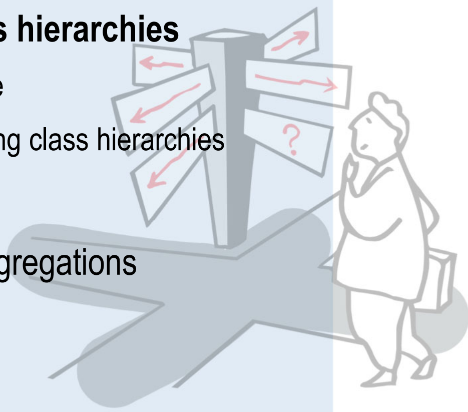
This may not always work… but let's see!

11

11

Roadmap

- Recap Lecture 5
- **Building good class hierarchies**
  - ☐ Types of inheritance
  - ☐ Heuristics for creating class hierarchies
- Roles and Protocols
- Associations and Aggregations
- Design by Contract

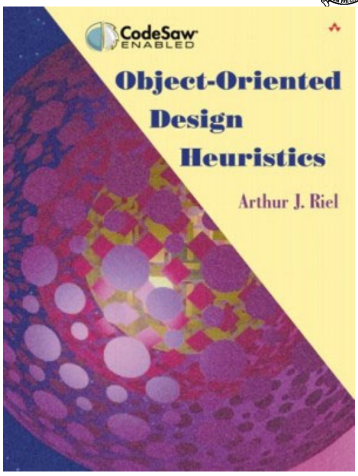


12

12

Object-Oriented Design Heuristics

- Seminal work, describing app. 60 heuristics to improve object-oriented design and implementation
- A selected few of the heuristics will be introduced in this lecture.



13

13

Heuristics for System Classes

- Heuristic 2.8 (p. 19)
  - *A class should capture one and only one key abstraction*
  - Key abstraction = important concept of the domain model
  - If this is violated, we immediately reduce **cohesion**
- Heuristic 2.9 (p. 20)
  - *Keep related data and behaviour in one place*
  - Otherwise, need to define (unnecessary) accessor methods and/or “data-holder” only classes, implying *increased* **coupling**

14

14

Heuristics for System Classes (cont.)

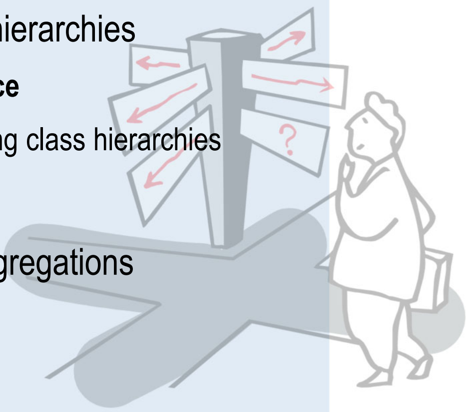
- Heuristic 3.1 (p. 33)
  - *Distribute system intelligence horizontally as uniformly as possible, that is, top-level classes of the system* should *share their work uniformly*
  - ☞ One needs to think where the system objects are created!
- Heuristic 3.2 (p. 33)
  - *Do not create god classes/objects in your system*
  - ☞ God class implies many responsibilities, hence significant complexity, hence difficult to understand and maintain.
  - ☞ God classes are often prone to contain many defects!

15

15

Roadmap

- Recap Lecture 5
- Building good class hierarchies
  - **Types of inheritance**
  - Heuristics for creating class hierarchies
- Roles and Protocols
- Associations and Aggregations
- Design by Contract



16

16

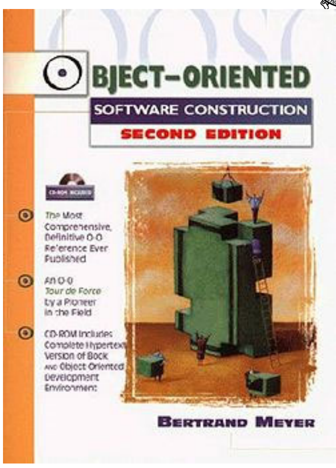
Inheritance = Incremental Derivation

17

17

Object-Oriented Software Construction

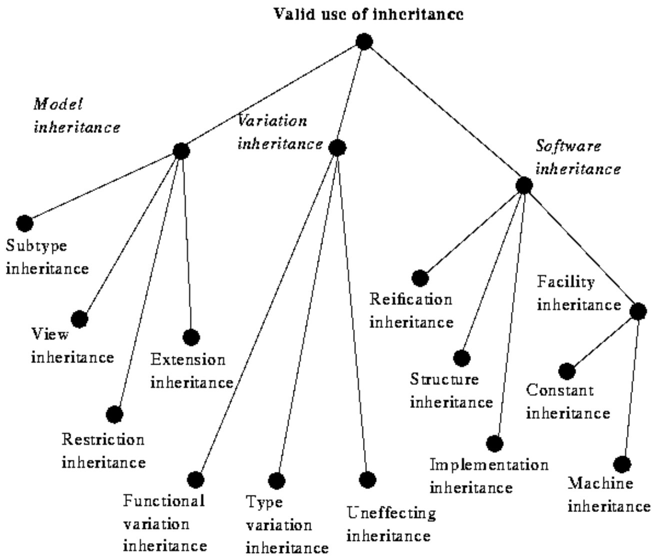
- Seminal book by Bertrand Meyer about foundations and practices of object-oriented design
- Contains a number of principles for object design
- Discusses "Design by Contract"



18

18

Taxonomy of Inheritance



19

Source: B. Meyer, Object-Oriented Software Construction, 1997

19

Taxonomy of Inheritance (cont.)

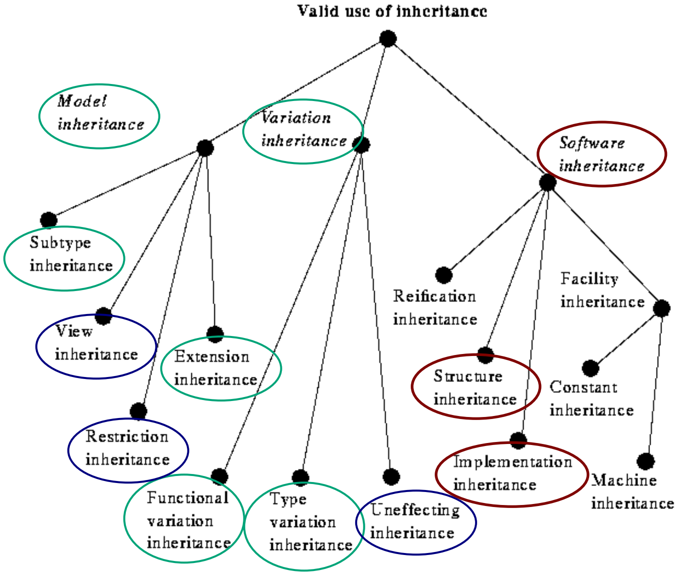
- *Model inheritance*, reflecting "is-a" relations between abstractions in the model.
- *Software inheritance*, expressing relations within the software itself rather than in the model.
  - Some of them are considered "bad practice"…
    - Derive a `Stack` from a `List` class
- *Variation inheritance* - a special case that may pertain either to the software or to the model - serving to describe a class through its differences with another class.

(cf. http://archive.eiffel.com/doc/manuals/technology/oosc/inheritance-design)

20

20

Taxonomy of Inheritance (cont.)



21

21

Heuristics for Inheritance

- Heuristic 5.1 (p. 81)
  - *Inheritance should be used only to model a specialization hierarchy*
  - ☞ We will revisit this heuristics later on…
- Heuristic 5.2 (p. 81)
  - *Derived classes must have knowledge of their base class (by definition), but base classes should not know anything about their derived classes*
  - ☞ Never make explicit reference to a subclass!!

22

22

Abstract Classes

Abstract classes factor out *common behavior* shared by other classes (i.e. their subclasses)

- group related classes with common responsibilities
- introduce abstract parent classes to represent the group
- ☞ "categories" are good candidates for abstract classes

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

23

23

Heuristics for Abstract Classes

- Heuristic 5.6 (p. 89)
  - ☐ *All abstract classes must be base classes*
  - ☞ No point having abstract classes that are not derived from!
- Heuristic 5.7 (p.89)
  - ☐ *All base classes should be abstract classes*
  - ☞ In an inheritance tree, only leaf nodes should be concrete classes, all other nodes are abstract.
  - ☞ Note: there is often a *perceived* need to violate this!

24

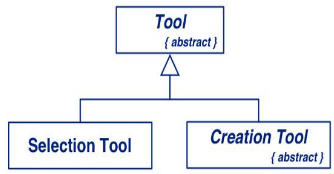
24

Sharing Responsibilities

[Concrete classes](#) may be both instantiated and inherited from.

☞ *Bad practice to subclass a concrete class, though!*

[Abstract classes](#) may only be inherited from.



Venn Diagrams can be used to visualize shared responsibilities.

(Warning: not part of UML!)

25

25

Building Good Hierarchies

Model a "kind-of" hierarchy:

- Subclasses should/must *support all inherited responsibilities*, and possibly more.
  - ☞ Subclasses may *refine/specialize* an inherited responsibility, but should *never hide* a responsibility from a parent class.

26

26

Heuristics for Base Classes

Factor common responsibilities as high as possible:

- Classes that *share common responsibilities* should *inherit from a common abstract superclass*; introduce any that are missing.
- Heuristic 5.8 (p. 93)
  - *Factor the commonality of data, behaviour and/or interface as high as possible in the inheritance hierarchy*
  - Reduces the amount of classes where specific information is kept
    - Reduces maintenance overhead later on
    - But: may introduce complexity early on…

27

27

Building Good Hierarchies (cont.)

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

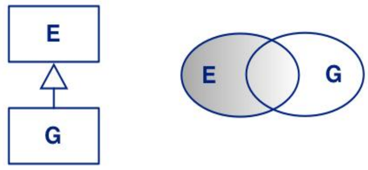
28

28

Building Kind-Of Hierarchies

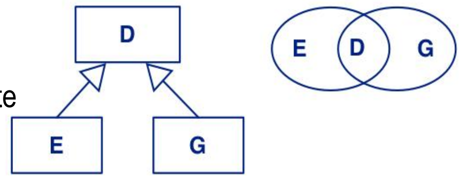
Incorrect Subclass/Superclass Relationships

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



Revised Inheritance Relationships

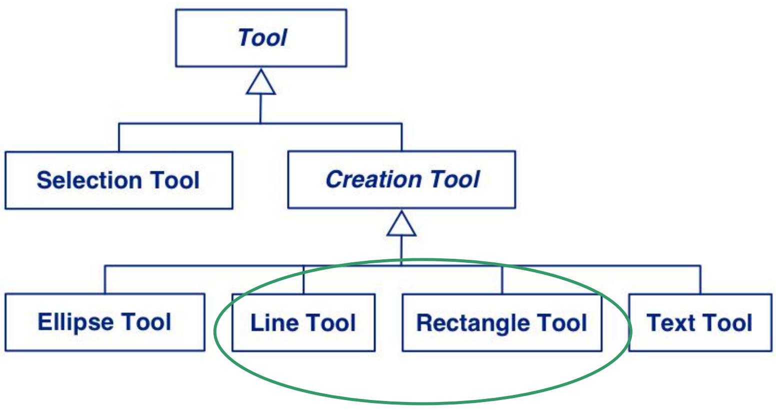
- Introduce *abstract superclasses* to encapsulate common responsibilities



29

29

Graphics Editor - Tool Hierarchy

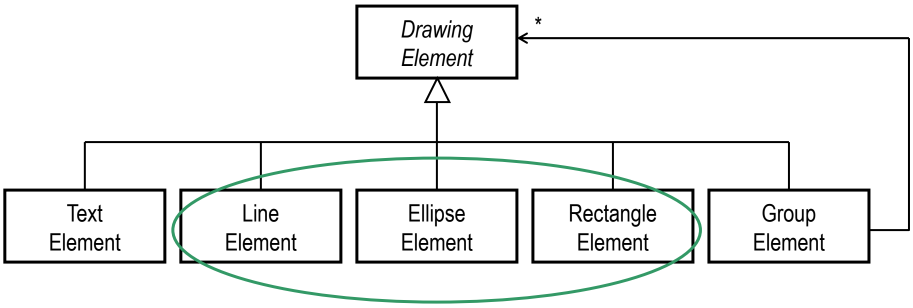


There is potential for additional shared behaviour
☞ may need to be further considered…

30

30

Graphics Editor - Element Hierarchy

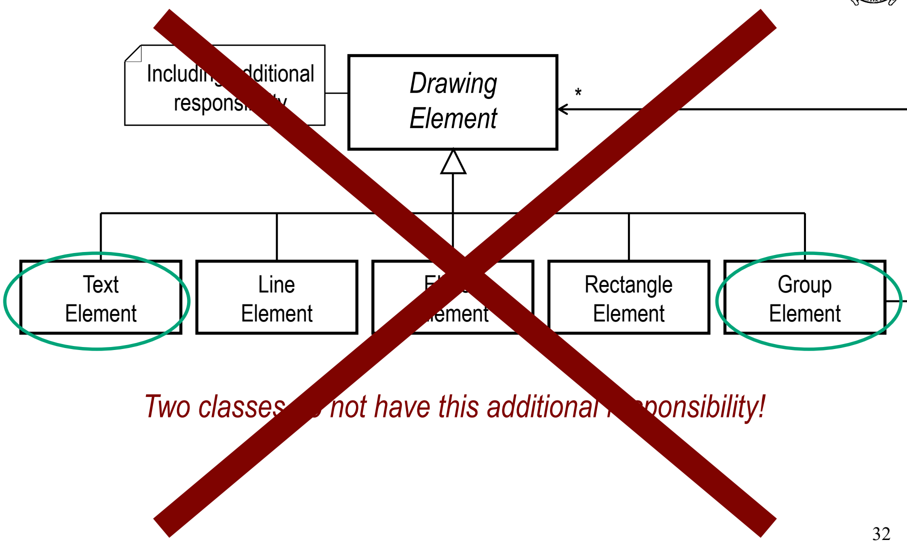


They all are drawn using one (or more) lines
☞ *common responsibility!*

31

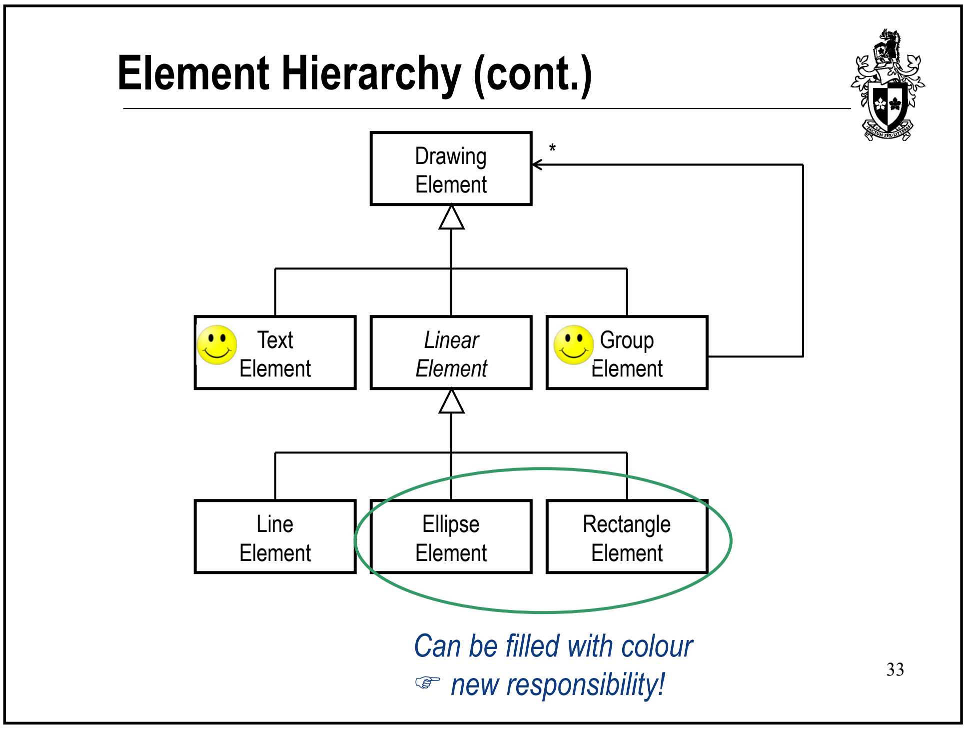
31

Element Hierarchy (cont.)

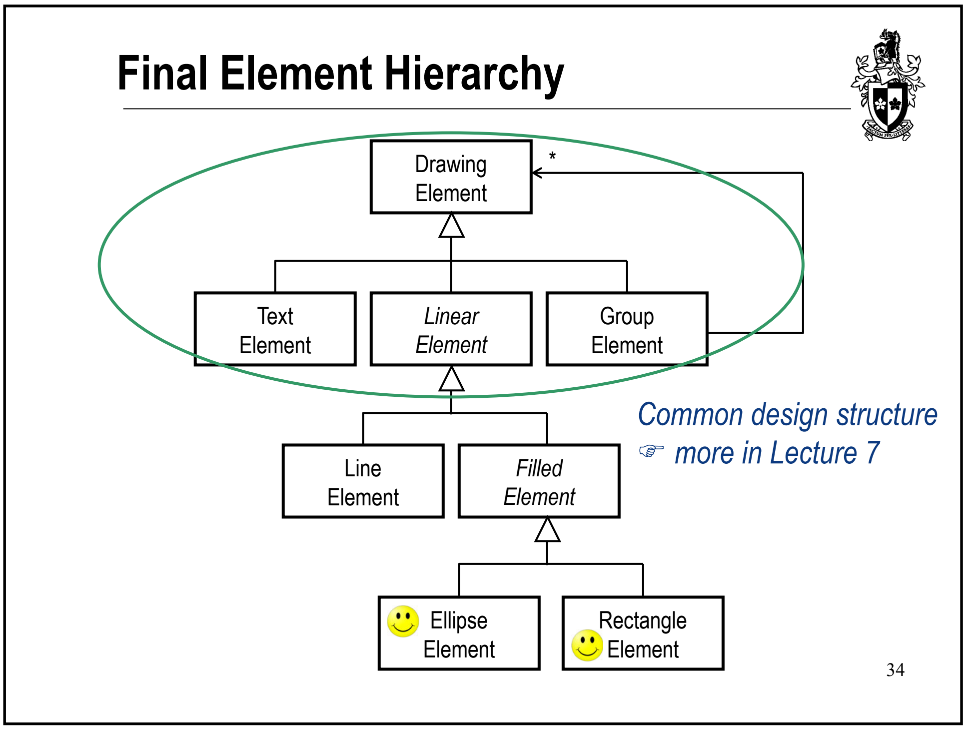


32

32


Element Hierarchy (cont.)
Drawing Element
*
Text Element
Linear Element
Group Element
Line Element
Ellipse Element
Rectangle Element
Can be filled with colour
☞ new responsibility!
33

33


Final Element Hierarchy
Drawing Element
*
Text Element
Linear Element
Group Element
Common design structure
☞ more in Lecture 7
Line Element
Filled Element
Ellipse Element
Rectangle Element
34

34

Inheritance Heuristics

- Heuristic 5.4 (p. 84)
  - *In theory, inheritance hierarchies should be deep - the deeper, the better*
  - ☞ “Correct” set of responsibilities at each level.
- Heuristic 5.5 (p. 84)
  - *In practice, inheritance hierarchies should be no deeper than an average developer can keep in his/her short-term memory*
  - ☞ Developers must be able to work with a given abstraction hierarchy! Assume 5 to 7 be a maximum depth.
  - ☞ Empirical studies show that hierarchies are rarely deep

35

35

Inheritance Heuristics (cont.)

- Heuristic 5.12 (p. 98)
  - *Explicit case analysis on the type of an object is usually an error. The designer should use polymorphism in most cases*
  - ☞ Case analysis also often reveals misassigned responsibilities!
- Heuristic 5.16 (p. 113)
  - *It should be illegal for a derived class to override a base class method with a method that does nothing.*
  - ☞ Declaring a NOP method implies that this class does not have a particular responsibility!

36

36

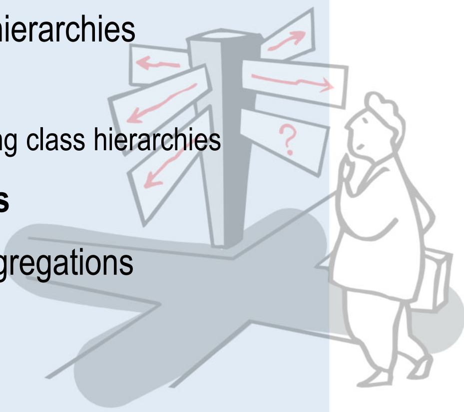
Break

37

37

Roadmap

- Recap Lecture 5
- Building good class hierarchies
  - ☐ Types of inheritance
  - ☐ Heuristics for creating class hierarchies
- **Roles and Protocols**
- Associations and Aggregations
- Design by Contract



38

38

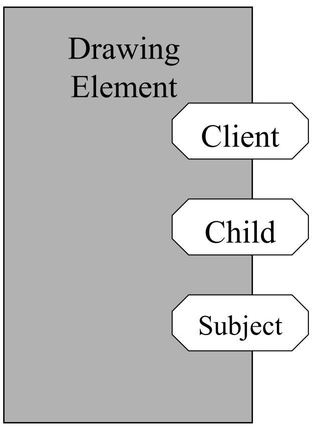
Roles and Protocols

- Within an object-oriented application, objects *collaborate* to perform the desired/required responsibilities.
- Collaborations are often described using *protocols* (interfaces and *interaction sequences/dependencies*)
- Each participant in a collaboration plays a given *role*.
- Roles can be either fixed or *unbound*.
  - Unbound roles can be used for tailoring an application (i.e. create a new class which can play this role).

39

39

Protocols and Roles (cont.)



Drawing Element is a class of the graphics editor; it defines three roles:

- *Client*: uses other elements of the application (i.e. other element classes; e.g. Line Element)
- *Child*: can be contained in a Group Element
- *Subject*: state being observed by other elements (e.g., Selection Tool)

Each of these roles implies

- a number of *related responsibilities*
- the usage of a particular *collaboration protocol* (e.g., a sequence of method invocations)

40

40

Interaction Heuristics

- Heuristic 4.1 (p. 57)
  - *Minimize the number of classes with which another class collaborates*
  - ☞ Too many collaborators generally implies misassigned and/or incorrectly identified responsibilities
- Heuristic 4.3 (p. 59)
  - *Minimize the amount of collaboration between a class and its collaborator(s), that is, the number of messages sent*
  - ☞ Implies that a collaboration protocol will have to be implemented using “few” methods
    - ☞ This can be quite problematic in some cases!

41

41

Interaction Heuristics (cont.)

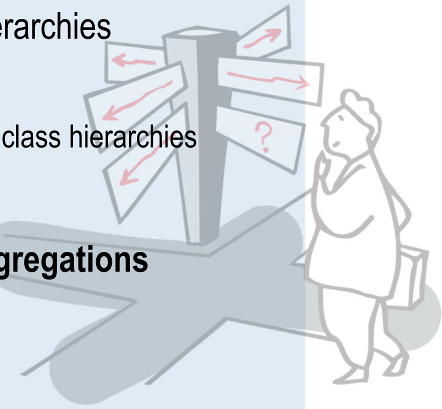
- Heuristic 3.9 (p. 43)
  - *Do not turn an operation into a class*
  - ☞ Be cautious when the name of a potential class is derived from a verb
  - ☞ However, there are situation when you do want operations to be encoded in a separate class!
    - ☞ For example, Strategies, Commands etc.
  - ☞ We will talk about those situations in the next lecture(s)…

42

42

Roadmap

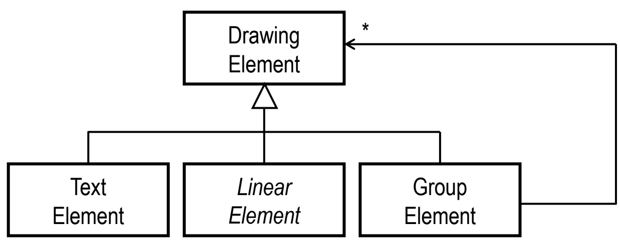
- Recap Lecture 5
- Building good class hierarchies
  - Types of inheritance
  - Heuristics for creating class hierarchies
- Roles and Protocols
- **Associations and Aggregations**
- Design by Contract



43

43

Associations and Aggregations



A `Group Element` contains multiple `Drawing Elements` (in fact, instances of concrete subclasses thereof)

- do we use aggregation or association to model this relationship?
- what is the “best” way to represent associations and/or aggregations?

44

44

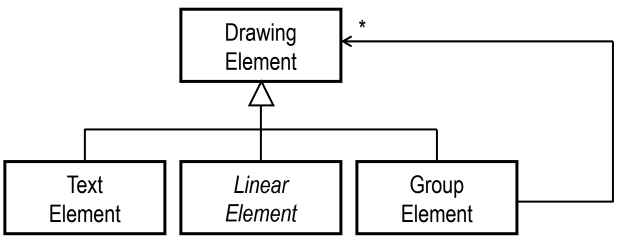
Heuristics for Associations

- Heuristic 7.1 (p. 147)
  - *When given a choice in an object-oriented design between a containment relationship and an association relationship, choose the containment relationship*
  - ☞ Reduces the amount of exposure of internals of a class
  - ☞ However, one needs to be careful to decide when containment (aggregation) reflects the given model and when not ("life of objects" rule).

45

45

Associations and Aggregations (cont.)



Clearly, the "children" of a `Group Element` can exist without its enclosing `Group Element` instance

☞ we need to use *association*, not aggregation, to model this relationship!

46

46

Collections

The elements of an association/aggregation generally need some form of expandable container/collection in the corresponding enclosing object

☞ *But what kind of container/collection do we choose?*

☞ We need to be clear what *kind of access* we need for the elements of an association/aggregation!

47

47

Collections (cont.)

- Only enumeration of elements needed
  - Choose the most general collection available
- Want to avoid duplicates?
  - Choose a set-like collection
- Need to be able to access specific elements?
  - Choose an "indexable" collection (e.g., Array, List, Hash)
- Need a specific order in enumerations
  - Choose a collection that allows ordering of elements
- Need some combination of the above?
  - Choose a collection that combines various behaviours

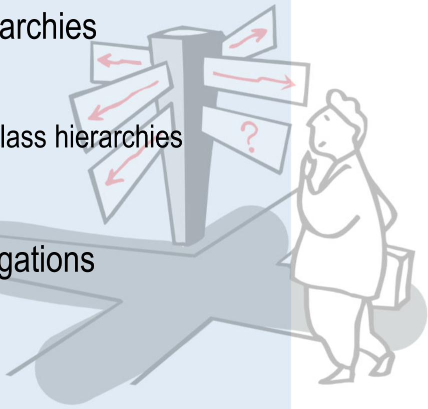
☞ *Most collection libraries provide various kinds of suitable classes*

48

48

Roadmap

- Recap Lecture 5
- Building good class hierarchies
  - Types of inheritance
  - Heuristics for creating class hierarchies
- Roles and Protocols
- Associations and Aggregations
- **Design by Contract**



49

49

Abstract State vs. Concrete State

- Every object (bar a few rare exceptions) contains statefull information (*an object has state*).
- Externally visible properties of this state defines its *abstract state*
  - Stack: size, order of elements, top element
- Internal representations define the *concrete state*
  - Stack may use a list or array to “store” its elements
- Clients should only ever rely on an object’s abstract state, not its concrete state.
- Object equality should only ever be defined over the abstract, but not concrete state.

50

50

Command-Query Separation Principle

Divide all methods of a class into two, non-overlapping categories:

- Queries
  - ☐ Return a result, but do not change the observable state of an object (that is, the method is free of *observable* side-effects)
  - ☞ *The abstract state of an object remains invariant!*
  - ☞ *However, the concrete state may change (e.g., for caching)*
- Commands (aka modifiers)
  - ☐ Change the (abstract) state of an object, but do not return a value (other than termination)

(cf. http://martinfowler.com/bliki/CommandQuerySeparation.html)

51

51

Design by Contract

- A design concept introduced by Bertrand Meyer in the late 1980s.
- Key premise: view the relationship between a service provider (e.g., an object) and service consumer (e.g., a client of this object – possibly another object) as a *formal agreement* (or *contract*), specifying each party's rights and obligations.
- In practice: extend the definition of abstract data types (e.g., classes) with *pre-conditions*, *post-conditions*, *invariants*, and how these three concepts can be refined in subclasses.

52

52

Example - Stack Abstraction

```
deferred class STACK [G]
feature -- Element change
    push (v : G) is            -- Push v onto top
            deferred
            require
                    not_full: size <= max_size
            ensure
                    item_pushed: top = v
                    size_increased: size = old size + 1
            end;
    pop is                     -- Remove top element
            deferred
            require
                    not_empty: size > 0
            ensure
                    item_removed: size = old size - 1
                    not_full: size <= max_size
            end;
    top : G is                 -- Get top element
            deferred
            require
                    not_empty: size > 0
            ensure
                    size_invariant: size = old size
            end;
    size : int is             -- Get the size of the stack
            deferred
            end;
end -- class Stack
```

Preconditions

Postconditions

53

53

Design by Contract (cont.)

- A *service consumer must ensure* that a service (method) on a provider is only ever invoked when this service's *pre-condition is met*.
  - Pre-conditions must be expressed in terms of the public interface of a service!
  - If the pre-condition is not met, there is no obligation on the provider's side to do anything "sensible"
- A *service provider must ensure* that, after successful completion of a service, the service *post-condition is met*.
  - A service consumer can rely on the specified post-condition.
  - Any errors in the execution of a service must be flagged to the consumer!

54

54

Design by Contract (cont.)

- Invariants of the service provider must hold after the *successful* execution of any of its externally visible services
  - During execution, they may be (obviously) violated…
- Subclasses can only
  - Weaken pre-conditions to services
  - Strengthen post-conditions of services and/or invariants
- Tools can assist in enforcing contracts and/or detect contract violations!

55

55

Benefits of Design by Contract

- Explicit notion of "contextualized" correctness
- Enforces separation of concerns
  - Helps in assigning responsibilities to the "correct" class
- Facilitates testing and debugging
  - There are even *smart compilers* that use it for optimizations!
- Enhances documentation
  - As a consequence, facilitates reuse

56

56

Summary

- Object-oriented decomposition is *not a mechanical task*
  - requires experience and a “bag full of best practices”
- Heuristics and Principles give you a *set of guidelines* (only)
  - *they do not define an algorithm for object design!*
- Apply (informed) *common sense* when you devise and restructure your design
  - common sense often leads to more comprehensible designs than any rules
- Early *verification* helps in getting your design “right”

57

57

Question for Review

1. What is the *Dependency Inversion Principle*?
2. How does Design by Contract help create reliable software? What are the downsides to this methodology?
3. In regards to OO design, how can abstract classes be used to improve code re-usability and portability?
4. Explain the effect of including a 'God Class' in a system built under OOD guidelines. In what ways does it undermine and reduce the effectiveness of OO design and its goals?
5. What is the difference between an abstract class and an interface? When should you use an interface over an abstract class?
6. What is a vapor class and how do they contribute to the design of a system? Are there instances where they could be useful?

58

58

“Exercise” from Lecture 4

Develop a domain model for a chess game that allows a human player to play a game of chess against a computer.

59

59

Question to Answer – week 6 (for week 7)

The spec of the “Question to Answer” is under the corresponding assignment setup, which will be released after the lecture.

60

60

Required Reading Lecture 7

Principal reading:

- Douglas C. Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann, *Inside Patterns* (available from Canvas)

Additional reading:

- Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, Chapter 1.
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns,* Chapter 1.
- Various articles from Doug Schmidt's Patterns page: http://www.dre.vanderbilt.edu/~schmidt/patterns-frameworks.html

61

61