

### 4.3 Guidelines for design

**Objectives** With regard to actual software development, it is important to develop a style of class design that avoids errors and improves maintenance and reuse. In this section a number of metrics are introduced that may be used to establish quantitative measures of object-oriented design. Also formal evaluation criteria for these metrics are discussed, which give a guideline for the interpretation of the outcome of applying the metrics to actual designs. The Law of Demeter provides a rule by which to organize and reduce dependencies between classes. Its underlying intuition is to avoid the visibility of objects embedded in other objects. In addition, we will discuss some guidelines for individual class design.

#### Points to emphasize

- *metrics* – objects, attributes, communication
- *the Law of Demeter* – ignorance is bliss
- *guidelines* – individual class design

**Hints** The formal definition of the metrics and the proofs concerning their behavior under the evaluation criteria given are in itself not very difficult, but may (with respect to the experience of your audience) be somewhat hard to swallow at first sight.

#### Questions

- (7) What metrics can you think of for object-oriented design? What is the intuition underlying these metrics?
- (8) What evaluation criteria for metrics can you think of? Are these sufficient for applying such metrics in actual software projects? Explain.
- (9) Give a formal definition of the following metric: WMC, DIT, NOC, CBO, RFC and LOC. Explain their meaning from a software engineering viewpoint.
- (10) What would be your rendering of the Law of Demeter? Can you phrase its underlying intuition? Explain.
- (11) Define the notions of client, supplier and acquaintance. What restrictions must be satisfied to speak of a preferred acquaintance, and a preferred supplier?

**Comments** This section must be regarded as a starting point for further exploration and research. One can think of incorporating metrics in a browsing tool. More importantly, however, are empirical results with regard to the validity of these metrics.

You may consult Lieberherr and Holland (1989) for a more detailed treatment and more examples of class transformations. You may also think of additional examples to convince students of the fact that *ignorance is bliss*. Is that a fact?

**Development process** – *cognitive factors* 4-20

- model → realize → refine

**Design criteria** – natural, flexible

- abstraction – *types*
- modularity – *strong cohesion* (class)
- structure – *subtyping*
- information hiding – *narrow interfaces*
- complexity – *weak coupling*

## **A metric suite**

4-21

- *WMC* – weighted methods per class
- *DIN* – depth of inheritance
- *NOC* – number of children
- *CBO* – coupling between objects
- *RFC* – response for a class
- *LCO* – lack of cohesion

## **Object-oriented design**

- *object definition* – *WMC*, *DIN*, *NOC*
- *attributes* – *RFC*, *LCO*
- *communication* – *RFC*, *CBO*

**Definitions**

4-22

- object names  $x, y$
- $iv(x) = \{i \mid i \text{ is variable of } x\}$
- $methods(x) = \{m \mid m \text{ is a method of } x\}$
- $properties(x) = iv(x) \cup methods(x)$

**Read/write properties**

- $iv(m_x) = \{i \mid m_x \text{ reads or writes } i\}$
- $methods(i_x) = \{m_x \mid i_x \in iv(m_x)\}$

**Cardinality**

- $| S | = \text{the cardinality of the set } S$

**Evaluation criteria**

4-23

- (i) non-coarseness —  
 $\exists x \exists y \bullet \mu(x) \neq \mu(y)$
- (ii) non-uniqueness = —  $\exists x \exists y \bullet \mu(x) = \mu(y)$
- (iii) permutation = —  $\exists x \exists y \bullet y$  is a permutation of  $x \wedge \mu(x) \neq \mu(y)$
- (iv) implementation = —  $\forall x \forall y \bullet \text{fun}(x) = \text{fun}(y) \not\Rightarrow \mu(x) = \mu(y)$
- (v) monotonicity = —  $\forall x \forall y \bullet \mu(x) \leq \mu(x + y) \ \& \ \mu(y) \leq \mu(x + y)$
- (vi) interaction = —  $\forall x \forall y \exists z \bullet \mu(x) = \mu(y) \wedge \mu(x + z) \neq \mu(y + z)$
- (vii) combination = —  $\exists x \exists y \bullet \mu(x) + \mu(y) < \mu(x + y)$

**Weighted methods per class**

WMC

4-24

**Measure** – *complexity of an object*

- $WMC(x) = \sum_{m \in methods(x)} complexity(m)$

**Viewpoint** – the number of methods and the complexity of methods is an indicator of how much time and effort is required to develop and maintain objects

**Depth of inheritance**

DIN

4-25

**Measure** – *scope of properties*

- $DIN(x) = distance(root(x), class(x))$

**Viewpoint** – the deeper a class is in the hierarchy, the greater the number of methods that is likely to be inherited, making the object more complex

**Number of children**

NOC

4-26

**Measure** – *scope of properties*

- $NOC(x) = | \text{descendants}(x) |$

**Viewpoint** – generally, it is better to have depth than breadth in the class hierarchy, since it promotes the reuse of methods through inheritance

**Coupling between objects**

CBO

4-27

**Measure** – *degree of dependence*

- $CBO(x) = | \{y \mid x \text{ uses } y \vee y \text{ uses } x\} |$

**Viewpoint** – excessive coupling between objects outside of the inheritance hierarchy is detrimental to modular design and prevents reuse

**Response for a class**

RFC

4-28

**Measure** – *complexity of communication*

- $RFC(x) = | \text{methods}(x) \cup \{m_y \mid x \text{ calls } m_y\} |$

**Viewpoint** – if a large number of methods can be invoked in response to a message, the testing and debugging of the object becomes more complex

**Lack of cohesion**

LCO

4-29

**Measure** – *similarity between methods*

- $LCO(x) = | \text{partitions}(\text{methods}(x), iv(x)) |$

where

- $\text{partitions}(M, I) = \{J \subseteq I \mid \text{methods}(J) \cap \text{methods}(I \setminus J) = \emptyset\}$

**Viewpoint** – cohesiveness of methods within a class is desirable since it promotes the encapsulation of objects

**Good Object-Oriented Design**

4-30

- *organize and reduce dependencies between classes*

**Client** – A method  $m$  is a client of  $C$  if  $m$  calls a method of  $C$

**Supplier** – If  $m$  is a client of  $C$  then  $C$  is a supplier of  $m$

**Acquaintance** –  $C$  is an acquaintance of  $m$  if  $C$  is a supplier of  $m$  but not (the type of) an *argument* of  $m$  or (of) an *instance variable* of the object of  $m$

□  $C$  is a *preferred acquaintance* of  $m$  if an object of  $C$  is created in  $m$  or  $C$  is the type of a global variable

□  $C$  is a *preferred supplier* of  $m$  if  $C$  is a supplier and  $C$  is (the type of) an instance variable, an argument or a preferred acquaintance

**Law of Demeter** – *ignorance is bliss*

4-31

Do not refer to a class  $C$  in a method  $m$  unless  $C$  is (the type of)

1. an instance variable
2. an argument of  $m$
3. an *object* created in  $m$
4. a global variable

□ *Minimize the number of acquaintances*

**Class transformations**

- *lifting* – make structure of the class invisible
- *pushing* – push down responsibility

### **Class design** – *guidelines*

4-32

- only methods public – *information hiding*
- do not expose implementation details
- public members available to all classes – *strong cohesion*
- as few dependencies as possible – *weak coupling*
- explicit information passing
- root class should be abstract model – *abstraction*

## 4.4 Towards a formal approach

**Objectives** This section is meant both as a conclusion with respect to the previous and present chapter and as a forward reference to part III. It states that design must allow for reasoning about a specification and the code that realizes a specification. Also, it must indicate how runtime consistency may be monitored by including appropriate conditions and invariance checks.

### Points to emphasize

- *contracts* – formal specification
- *verification* – as a design methodology
- *runtime consistency* – invariance

**Hints** You may include additional material on formal methods.

### Question

(12) Characterize the elements that form part of a formal specification.

**Comments** It is my experience that students are quite interested in your opinion on software engineering issues in general, and the question of whether a formal approach is useful in particular. Don't hesitate to express your opinion!