Advanced Object-Oriented Design
Lecture 4

Object-oriented metrics

Bartosz Walter
<Bartek.Walter@man.poznan.pl>

Agenda

1. Motivation and goals
2. Complexity metrics
3. Metrics for Object-Oriented Design (MOOD suite)
4. Metrics suite by Chidamber-Kemerer
5. Metrics by R. Martin
6. Law of Demeter

Agenda

Goals

Goals for OO metrics

Why OO metrics?

- the instrumentarium changes: localization, encapsulation, information hiding, inheritance, and object abstraction techniques
- they are supposed to be better suited for measuring OO systems than traditional ones
- they are better related to external attributes of OO systems, like fault-proneness or maintainability

Agenda

Complexity metrics

Cyclomatic Complexity (McCabe, 1976)

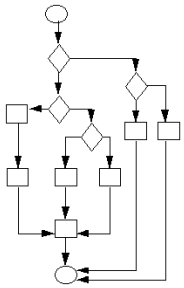
Idea
to evaluate the complexity of an algorithm

Definition
- the number of independent paths within a procedure (method)
- the count of test-cases needed to test the method comprehensively

Formula
$CC = \overline{E} - \overline{V} + 2$

Remarks
- lower *CC* **may** imply decreased testing and increased understandability or that decisions are deferred through message passing, but **not** that the method is not complex
- *CC* measures methods, not classes (due to inheritance), but combined together with other measures it may also evaluate the complexity of a class
- CC values greater than 20 suggest the method is too complex

Example



If G is the control flowgraph of program P and G has e edges (arcs) and n nodes

$v(P) = e-n+2$

$v(P)$ is the number of linearly independent paths in G

here $e = 16$ $n = 13$ $v(P) = 5$

More simply, if d is the number of decision nodes in G then

$v(P) = d+1$

source: http://www.dcs.qmw.ac.uk/~norman/papers/qa_metrics_article

Agenda

MOOD suite

Metrics for Object-Oriented Design (MOOD)

- **Defined in 1995 by F. B. e Abreu**
- **System-level**
- **Expressed as quotients (percentages) ranging from 0% to 100%**
  - numerator represents the actual use of one of those mechanisms for a given design
  - denominator, acting as a normalizer, represents the hypothetical maximum achievable use for the same mechanism on the same design
- **Dimensionless**
- **Independent of system size**
- **Make no reference to programming language**

Metrics for Object-Oriented Design (MOOD)

Refer to basic structural mechanisms of the object-oriented paradigm

- **Encapsulation**
  - Method Hiding Factor (MHF)
  - Attribute Hiding Factor (AHF)
- **Inheritance**
  - Method Inheritance Factor (MIF)
  - Attribute Inheritance Factor (AIF)
- **Polymorphism**
  - Polymorphism Factor (PF)
- **Message passing**
  - Coupling Factor (CF)

Attribute and Method Hiding Factor

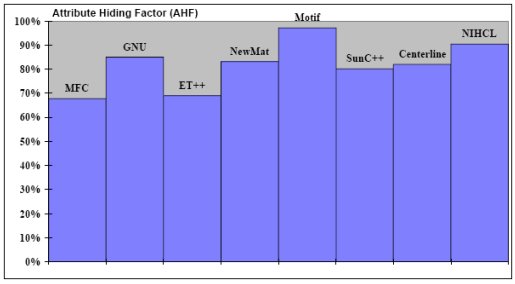
AHF and MHF are measures of the use of the information hiding concept that is supported by the encapsulation mechanism

$$AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)} = 1 - \frac{\sum_{i=1}^{TC} A_v(C_i)}{\sum_{i=1}^{TC} A_d(C_i)}$$

$$MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)} = 1 - \frac{\sum_{i=1}^{TC} M_v(C_i)}{\sum_{i=1}^{TC} M_d(C_i)}$$

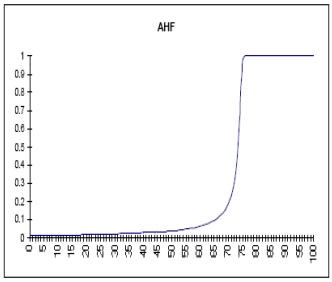
X_d – a member
X_h – hidden member
X_v – visible member
$X_d = X_h + X_v$

Attribute Hiding Factor



source: F.B. e Abreu, ECOOP 1995

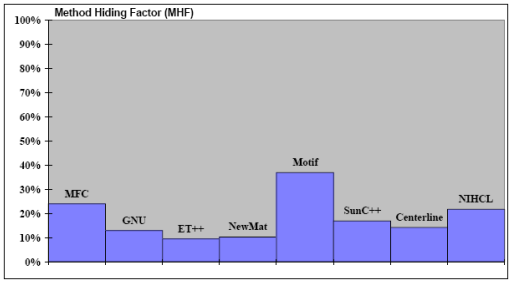
Attribute Hiding Factor



Attributes should be hidden as much as possible. Ideally all attributes would be hidden (being only accessed by the corresponding class methods)

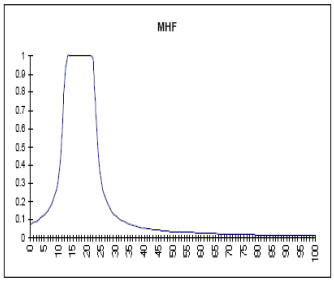
source: F.B. e Abreu, ECOOP 1995

Method Hiding Factor



source: F.B. e Abreu, ECOOP 1995

Method Hiding Factor

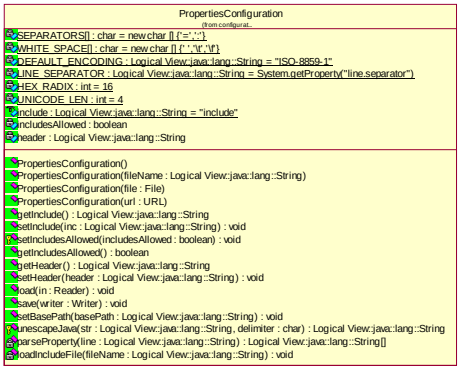


Very low MHF indicate an insufficiently abstracted implementation.

Very high MHF indicate very little functionality

source: F.B. e Abreu, ECOOP 1995

Example



MHF = ?

AHF = ?

http://jakarta.apache.org/commons/configuration

Attribute and Method Inheritance Factor

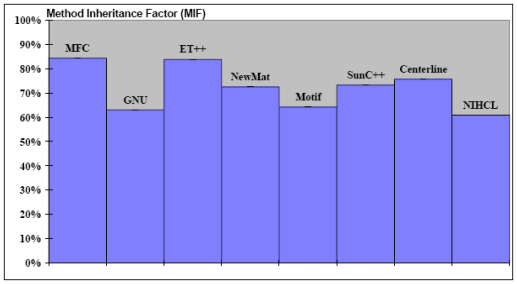
MIF and AIF are measures of inheritance. This allows (1) expressing similarity among classes, (2) the portrayal of generalization and specialization relations, (3) simplification of the definition of inheriting classes, by means of reuse.

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)} = 1 - \frac{\sum_{i=1}^{TC} M_d(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)} = 1 - \frac{\sum_{i=1}^{TC} A_d(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

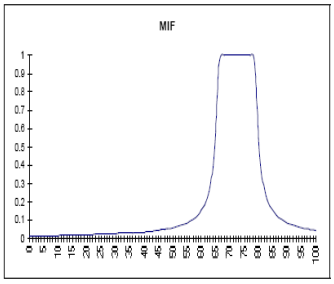
- X_a – available member
- X_d – defined member
- X_i – inherited member
- $X_a = X_d + X_i$
- X_n – new member
- X_o – overriden member
- $X_d = X_n + X_o$

Method Inheritance Factor



source: F.B. e Abreu, ECOOP 1995

Method Inheritance Factor

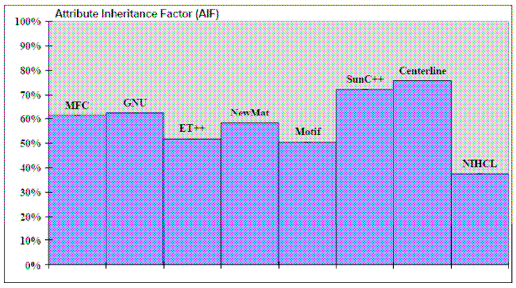


Low MIF indicate a low inheritance reuse ratio.

Very high MIF indicate a complicated inheritance tree

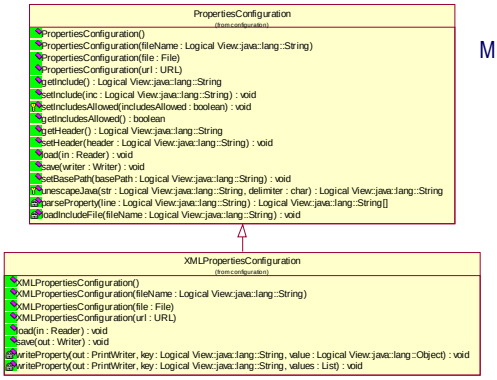
source: F.B. e Abreu, ECOOP 1995

Attribute Inheritance Factor



source: F.B. e Abreu, ECOOP 1995

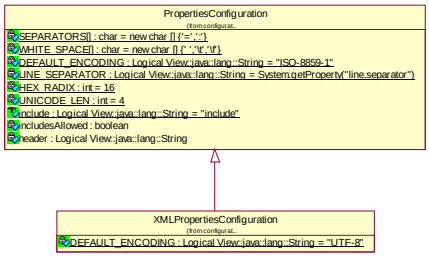
Example



MIF = ?

http://jakarta.apache.org/commons/configuration

Example



AIF = ?

http://jakarta.apache.org/commons/configuration

Polymorphism Factor

PF is a measure of polymorphic overriding. By allowing to bind a common message call to one of several class instances, polymorphism allows

- to build flexible systems
- refinement of the taxonomy without side-effects

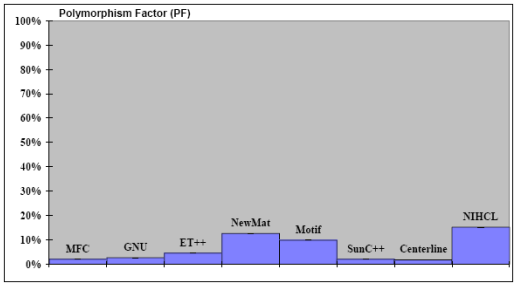
$$PF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} \left[M_n(C_i) \times DC(C_i) \right]}$$

M_o – overriden methods

M_n – new methods

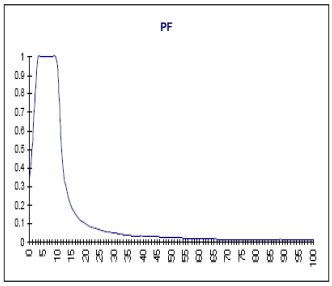
DC – descendant classes

Polymorphism Factor



source: F.B. e Abreu, ECOOP 1995

Polymorphism Factor

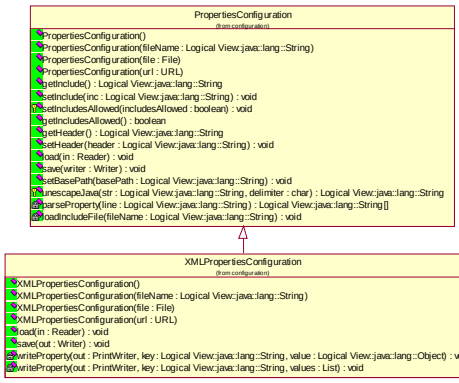


Very low PF indicate structural design.

High PF indicate a complicated inheritance tree

source: F.B. e Abreu, ECOOP 1995

Example



PF = ?

http://jakarta.apache.org/commons/configuration

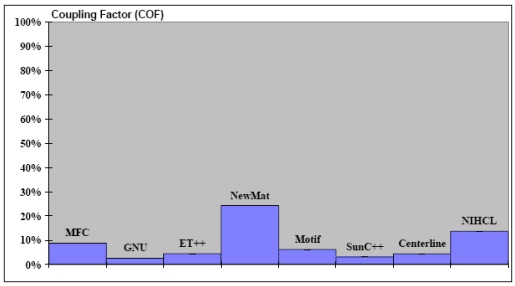
Coupling Factor

CF is a measure of dependability of individual classes on each other.

$$COF = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} is\_client(C_i, C_j)\right]}{TC^2 - TC - 2 \times \sum_{i=1}^{TC} DC(C_i)}$$

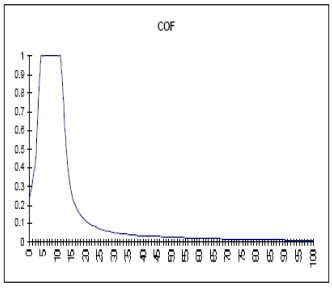
$$is\_client(C_c, C_s) = \begin{cases} 1 & iff \quad C_c \Rightarrow C_s \wedge C_c \neq C_s \\ & \wedge \neg(C_c \rightarrow C_s) \\ 0 & otherwise \end{cases}$$

Coupling Factor



source: F.B. e Abreu, ECOOP 1995

Coupling Factor

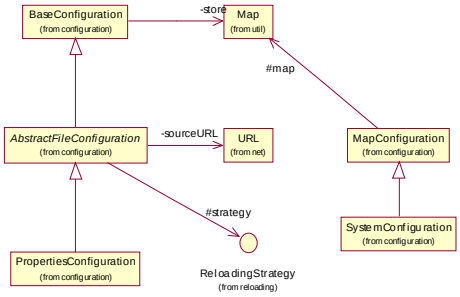


Low CF suggest low functionality, limited to few classes.

High CF indicate a poor design.

source: F.B. e Abreu, ECOOP 1995

Example



CF = ?

http://jakarta.apache.org/commons/configuration

Agenda

Metrics by Chidamber & Kemerer

Metrics suite by Chidamber-Kemerer

- **Defined by S. R. Chidamber i C. F. Kemerer in 1991**
- **Class-level**
- **Includes six metrics:**
  - Response For Class (RFC)
  - Weighted Method per Class (WMC)
  - Depth of Inheritance Tree (DIT)
  - Number Of Children (NOC)
  - Lack of Cohesion of Methods (LCOM)
  - Coupling Between Objects (CBO)

Response for Class

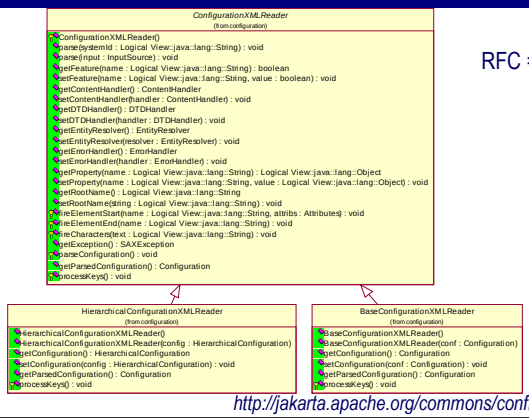
Idea
to measure potential communication between the class and other classes

Definition
the count of methods that can be invoked in response to a message sent to an object of the class

Formula
$RFC = \overline{M} + \overline{M}_{subclasses}$

Remarks
- A class with larger response is considered more complex
- High *RFC* suggests that testing and debugging of the class becomes complicated

Example



RFC = ?

http://jakarta.apache.org/commons/configuration

Weighted Methods per Class

Idea
to measure the complexity of a class

Definition
- the count of methods implemented within a class (unweighted)
- or the sum of complexities of its methods (weighted)

Formula
$WMC = \overline{M}$, or
$WMC = \Sigma\ CC_M$

Remarks
- *WMC* predicts of how much time and effort is required to develop and maintain the class
- classes with high *WMC* are more specific, thus reducing reuse

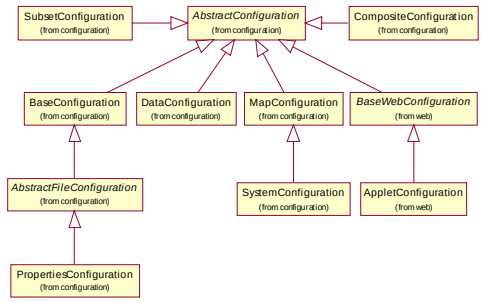
Depth of Inheritance Tree

Idea
to measure complexity of inheritance-related hierarchies

Definition
- *DIT* is the maximum number of steps from the class node to the root of the tree and is measured by the number of ancestor classes

Remarks
- The deeper a class is within the hierarchy, the greater the number methods it is likely to inherit making it more complex to predict its behavior.
- Deeper trees constitute greater design complexity, since more methods and classes are involved, but the greater the potential for reuse of inherited methods.

Example



DIT = ?

http://jakarta.apache.org/commons/configuration

Number of Children

Idea

to measure potential impact of modification in a class

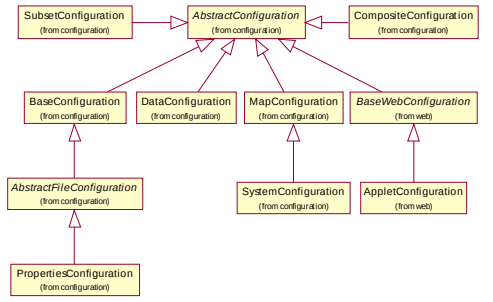
Definition

- *NOC* is the number of immediate subclasses (implementations) of the class

Remarks

- The greater *NOC*, the greater the likelihood of improper abstraction of the parent and may be a case of misuse of subclassing.
- The greater *NOC*, the more testing the class demands.
- The greater *NOC*, the greater the reuse since inheritance is a form of reuse.

Example



NOC = ?

http://jakarta.apache.org/commons/configuration

Lack of Cohesion of Methods (Chidamber & Kemerer, 1993)

Idea

to measure dissimilarity of methods by instance variable or class attribute

Definition

Take each pair of methods in the class and determine the set of fields they each access. If the sets are disjoint, the count *P* increases by one. If they share at least one field access, *Q* increases by one. After considering each pair of methods:

Formula

$LCOM1 = (P > Q) ? (P - Q) : 0$

Remarks

- *LCOM1* gives value of 0 for different classes
- the definition is based on method-data interaction, which may not be a correct way to define cohesiveness in the object-oriented world
- *LCOM1* is defined on variable access, it's not well suited for classes that internally access their data via properties

Lack of Cohesion of Methods (Henderson-Sellers, 1996)

Idea

to measure dissimilarity of methods by instance variable or class attribute

Definition

m	a number of methods in a class
a	a number of attributes in a class
mA	a number of methods that access the attribute *A*
sum(mA)	sum of all *mA* over all the attributes in the class

Formula

$LCOM3 = (m - sum(mA) / a) / (m - 1)$

Remarks

- *LCOM3* values varies from 0 to 2
- *LCOM3* > 1 indicates lack of cohesion and the class should be split
- If ***m*** = 1 or ***a*** = 0, then *LCOM3* is undefined and displayed as 0

Example

```
public class FileChangedReloadingStrategy
implements ReloadingStrategy {
    private static final String JAR_PROTOCOL = "jar";
    private static final int DEFAULT_REFRESH_DELAY = 5000;
    protected FileConfiguration configuration;
    protected long lastModified;
    protected long lastChecked;
    protected long refreshDelay = DEFAULT_REFRESH_DELAY;

    public void setConfiguration(FileConfiguration conf) {
        this.configuration = conf;
    }

    public void init() {
        updateLastModified();
    }

    public boolean reloadingRequired() {
        boolean reloading = false;
        long now = System.currentTimeMillis();
        if (now > lastChecked + refreshDelay) {
            lastChecked = now;
            if (hasChanged())
                reloading = true;
        }
        return reloading;
    }

    public void reloadingPerformed(){
        updateLastModified();
    }

    public long getRefreshDelay(){
        return refreshDelay;
    }
```

```
    public void setRefreshDelay(long refreshDelay) {
        this.refreshDelay = refreshDelay;
    }

    protected void updateLastModified() {
        File file = getFile();
        if (file != null)
            lastModified = file.lastModified();
    }

    protected boolean hasChanged() {
        File file = getFile();
        if (file == null || !file.exists())
            return false;
        return file.lastModified() > lastModified;
    }

    protected File getFile() {
        return (configuration.getURL() != null) ?
                fileFromURL(configuration.getURL()) :
                configuration.getFile();
    }

    private File fileFromURL(URL url){
        if (JAR_PROTOCOL.equals(url.getProtocol())){
            String path = url.getPath();
            try {
                return ConfigurationUtils.fileFromURL(
                    new URL(path.substring(0,
                    path.indexOf('!'))));
            } catch (MalformedURLException mex) {
                return null;
            }
        } else {
            return ConfigurationUtils.fileFromURL(url);
        }
    }
}
```

LCOM = ?

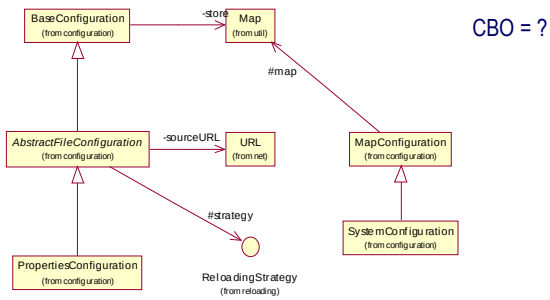
Coupling Between Objects

Idea
to measure class dependency on other non-ancestor classes

Definition
- *CBO* is the number of non-inherited classes associated with the target class
- It is counted as the number of types that are used in attributes, parameters, return types, throws clauses, etc.
- Primitive and basic system types (e.g. *java.lang.**) are not counted

Other coupling metrics
- *Data Abstraction Coupling* (*DAC*): the total number of referred types in attribute declarations. Primitive types, system types, and types inherited from the super classes are not counted.
- *Method Invocation Coupling* (*MIC*): the relative number of classes that receive messages from a particular class.

Example



CBO = ?

http://jakarta.apache.org/commons/configuration

Agenda

Metrics by R. C. Martin

Metrics suite by R. Martin

- **Defined by R. Martin in 1994**
- **Package and class-level**
- **Metrics consider dependency vs. stability trade-off**
- **Includes five metrics:**
  - Efferent Coupling (Ce)
  - Afferent Coupling (Ca)
  - Instability (I)
  - Abstractness (A)
  - Normalized Distance from Main Sequence (D)

Efferent Coupling

Idea
to measure the given module's dependency (incoming dependency) on external modules

Definition
- *Ce* is the number of classes inside a module that depend upon classes outside the module

Formula
Ce = number of types, on which the module depends

Remarks
- High *Ce* indirectly suggests module's instability (independence)
- Highly efferent modules have little responsibility to other packages, but reversely depend on them
- Preferred values range from 0 to 20
- Example: GUI components

Afferent Coupling

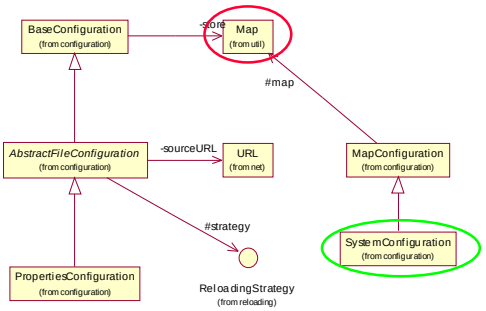
Idea
to measure the dependency of external modules (outgoing dependency) on the given module

Definition
- *Ca* is the number of classes and interfaces outside a module that depend upon classes and interfaces within the module

Formula
Ca = number of types, which depend on the module

Remarks
- High *Ca* indirectly suggests module stability (responsibility)
- Highly afferent packages bear large responsibility to other modules
- Difficult to change without affecting dependent modules
- Preferred values range from 0 to 500
- Example: bussiness objects, controllers

Example



http://jakarta.apache.org/commons/configuration

Instability

Idea
to measure package stability (dependency on other packages)

Definition
- *I* is the relation of efferent (outcoming) couplings to all couplings

Formula
$I = Ce / (Ce + Ca)$

Remarks
- Packages that contain multiple outgoing but few incoming dependencies (*I* is close to 1) are less stable because of the consequences of changes in these packages.
- Packages containing more incoming dependencies are more stable (*I* is close to 0) because they are more difficult to change.
- Designs of packages should intentionally be made as stable (0.0; 0.3) or unstable (0.7;1.0) as possible.

Abstractness

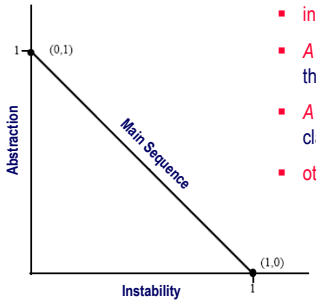
Idea
to measure the degree of how abstract data types are used

Definition
- The number of abstract classes (and interfaces) within a package divided by the total number of types in a package

Formula
$A = T_{Abstract} / (T_{Abstract} + T_{Concrete})$

Remarks
- Abstract classes are responsible (changes to them propagate to dependents) and stable

Main sequence



- in ideal case: $A + I = 1$
- $A = 0$ and $I = 1$: concrete classes that cannot be extended
- $A = 1$ and $I = 0$: purely abstract classes
- other: a trade-off between *A* and *I*

source: R. Martin, 1994

Normalized Distance from Main Sequence

Idea
to measure the balance between abstractness and stability

Definition
- *D* is a perpendicular distance from the ideal balance of I and A

Formula
$D = | A + I - 1 |$

Remarks
- Any class with *D* far from zero should be re-examined and restructured
- The metric is subject to statistical analysis

Agenda

Law of Demeter

Law of Demeter (Lieberherr & Holland, 1989)

Idea

- to restrict long message calls chains
- „only talk to your (immediate) friends"
- „never talk to strangers"
- an object should avoid invoking methods of a member object returned by another method

```
public class Customer {
  public Operation[] operationsAt(Date date) {
    Operation[] op = customer.getAccount().getHistory().getEntriesAt(aDate);
  }
}
```

Operation[] Customer Account History Operation[]

Law of Demeter

Definitions

Client: Method *M* is a client of method *N* of class *C*, if inside *M* message *N* is sent to an object of class *C*, or to *C*. If *N* is specialized in one or more subclasses, then *M* is only a client of *N* attached to the highest class in the hierarchy. Method *M* is a client of some method of class *C*.

Supplier: If *M* is a client of class *C* then *C* is a supplier to *M*.

Acquaintance Class: A class *C1* is an acquaintance class of method *M* of class *C2*, if *C1* is a supplier to *M* and *C1* is not one of the following:

- the same as *C2*;
- a class used in the **declaration of an argument** of *M*
- a class used in the **declaration of an instance variable** of *C2*

Preferred-supplier class: Class *B* is called a preferred-supplier to method *M* (of class *C*) if *B* is a supplier to *M* and one of the following conditions holds:

- *B* is used in the **declaration of an instance variable** of *C*
- *B* is used in the **declaration of an argument of *M***, including *C* and its superclasses
- *B* is a **preferred acquaintance** class of *M*.

Strict form of Law of Demeter

Strict form

every supplier class of a method must be a preferred supplier

Simply speaking

Every method *M* of object *O* may invoke only methods of following kinds of objects:

- itself,
- its parameters,
- any objects it creates/instantiates,
- its direct component objects.

Weak form of Law of Demeter

Weak form

every supplier class of a method must be a preferred supplier or its subclass

Simply speaking

Every method *M* of object *O* may invoke only methods of following kinds of objects:

- itself,
- its parameters or any subclass of them,
- any objects it creates/instantiates or any subclass of them,
- its direct component objects or any subclass of them.

Law of Demeter

Comments

- Resulting software tends to be more maintainable and adaptable
- Responsibility for accessing subparts is passed from the calling method to owning object
- LoD reduces coupling
- LoD enforces structure hiding (abstraction)
- LoD promotes type localization and narrowing interfaces
- LoD increases number of delegating methods in intermediate objects
- LoD has been experimentally confirmed to reduce probability of fault ratio (Basili, 1996)

Summary of object-oriented metrics

Src	Metric	Scope	Feature
MC	Cyclomatic Complexity (CC)	M	Complexity
?	Lines of Code (LOC)	M/C	Complexity
MO	Attribute/Method Hiding Factor (AHF/MHF)	S	Encapsulation
MO	Attribute/Method Inheritance Factor (AIF/MIF)	S	Inheritance
MO	Polymorphism Factor (PF)	S	Inheritance
MO	Coupling Factor (CF)	S	Dependency
CK	Weighted Method per Class (WMC)	C	Complexity
CK	Response for a Class (RFC)	C	Complexity
CK	Lack of Cohesion of Methods (LCOM)	C	Cohesion
CK	Coupling Between Objects (CBO)	C	Dependency
CK	Depth of Inheritance Tree (DIT)	C	Inheritance
CK	Number of Children (NOC)	C	Inheritance
M	Afferent Coupling (Ca)	P	Dependency
M	Efferent Coupling (Ce)	P	Dependency
M	Instability (I)	P	Dependency
M	Abstractness (A)	P	Dependency

Readings

1. S.R. Chidamber, C.F. Kemerer, *A metrics suite for object-oriented design.* IEEE Transactions on Software Engineering, Vol. 20, No 6, pp. 476-493
2. F. B. Abreu, *The MOOD Metrics Set.* ECOOP 1995 Workshop on Metrics.
3. R. Martin, *OO Design Quality Metrics. http://www.objectmentor.com/publications/oodmetrc.pdf*
4. *Applying and interpreting OO Metrics. NASA SATC, http://satc.gsfc.nasa.gov/support/STC_APR98/apply_oo/apply_oo.html*
5. *Eclipse Metrics Plugin. http://metrics.sourceforge.org*

Q&A