SOFTWARE DESIGN

Package design principles, Software metrics

Content

- Package Design
  - Cohesion Principles
  - Coupling Principles
- Software metrics

References

- David Patterson, Armando Fox, Engineering Long-Lasting Software: An Agile Approach Using SaaS and Cloud Computing, Alpha Ed.[Patterson]
- Taylor, R., Medvidovic, N., Dashofy, E., Software Architecture: Foundations, Theory, and Practice, 2010, Wiley [Taylor]
- Gillibrand, David, Liu, Kecheng. Quality Metric for Object-Oriented Design. Journal of Object-Oriented Programming. Jan 1998.
- Li, Wei. Another Metric Suite for Object–Oriented programming. Journal of Systems and Software. vol. 44, Feb. 1998
- ETHZ course materials
- Univ. of Aarhus Course Materials
- Univ. of Utrecht Course Materials

High-level Design

- Dealing with *large-scale systems*
  - > 50 KLOC
  - team of developers, rather than an individual
- Classes are a valuable but not sufficient mechanism
  - too *fine-grained* for organizing a large scale design
  - need mechanism that impose a higher level of order

Packages

- a logical grouping of declarations that can be imported in other programs
- containers for a group of classes (UML)
  - reason at a higher-level of abstraction

Issues of High-Level Design

Goal

- *partition* the classes in an application according to some *criteria* and then *allocate* those partitions to packages

Issues

- What are the best partitioning criteria?
- What principles govern the design of packages?
  - *creation* and *dependencies* between packages
- Design packages first? Or classes first?
  - i.e. *top-down* vs. *bottom-up approach*

Approach

- Define principles that govern package design
  - the creation and interrelationship and use of packages

Principles of OO High-Level Design

- Cohesion Principles
  - Reuse/Release Equivalency Principle (REP)
  - Common Reuse Principle (CRP)
  - Common Closure Principle (CCP)
- Coupling Principles
  - Acyclic Dependencies Principle (ADP)
  - Stable Dependencies Principle (SDP)
  - Stable Abstractions Principle (SAP)

What is really Reusability ?

- Does copy-paste mean reusability?
  - Disadvantage: **You** own that copy!
    - you must change it, fix bugs.
    - eventually the code diverges
  - Maintenance is a nightmare
- Martin's Definition:
  - *I reuse code if, and only if, I never need to look at the source-code*
  - treat reused code like a *product* ⇒ don't have to maintain it

- Clients (re-users) may decide on an appropriate time to use a newer version of a component release

Reuse/Release Equivalency Principle (REP)

- *The granule of reuse is the granule of release. Only components that are released through a tracking system can be efficiently reused.* [R. Martin]

- *Either all the classes in a package are reusable or none of it is!* [R. Martin]

What does this mean?

- Reused code = product
  - Released, named and maintained by the producer.
- Programmer = client
  - Doesn’t have to maintain reused code
  - Doesn’t have to name reused code
  - May choose to use an older release

The Common Reuse Principle

All classes in a package [library] should be reused together. If you reuse one of the classes in the package, you reuse them all. [R.Martin]

If I depend on a package, I want to depend on every class in that package! [R.Martin]

What does this mean?

- Criteria for grouping classes in a package:
  - Classes that tend to be reused together.
- Packages have physical representations (shared libraries, DLLs, assembly)
  - Changing just one class in the package -> rerelease the package -> revalidate the application that uses the package.

Common Closure Principle (CCP)

The classes in a package should be closed against the same kinds of changes.

A change that affects a package affects all the classes in that package

[R. Martin]

What does this mean?

- Another criteria of grouping classes:
  - Maintainability!
  - Classes that tend to change together for the same reasons
  - Classes highly dependent
- Related to OCP
  - How?

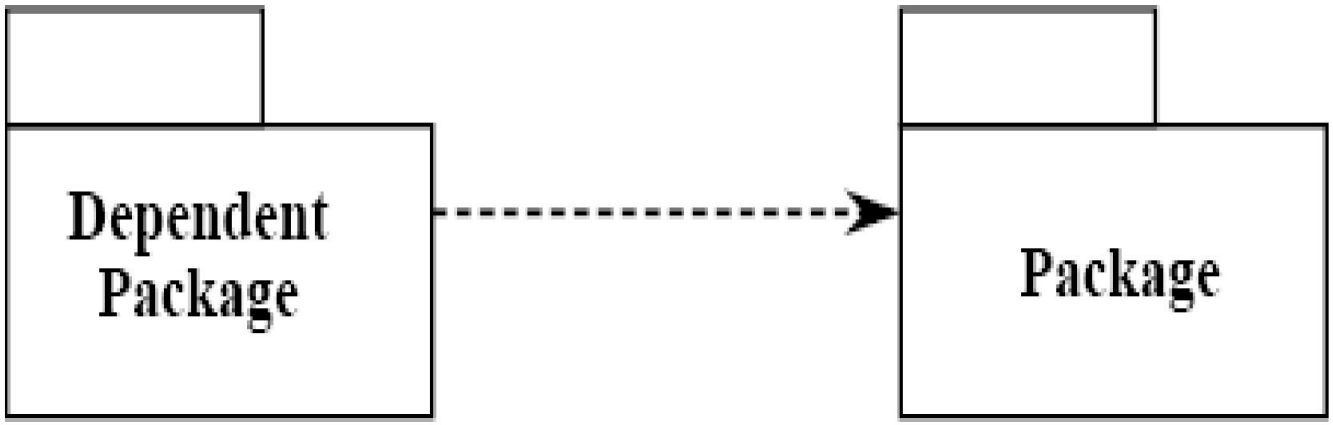
Reuse vs. Maintenance

- REP and CRP makes life easier for reuser
  - packages very small
- CCP makes life easier for maintainer
  - large packages
- Packages are not fixed in stone
  - early in project focus on CCP
  - later when architecture stabilizes: focus on REP and CRP

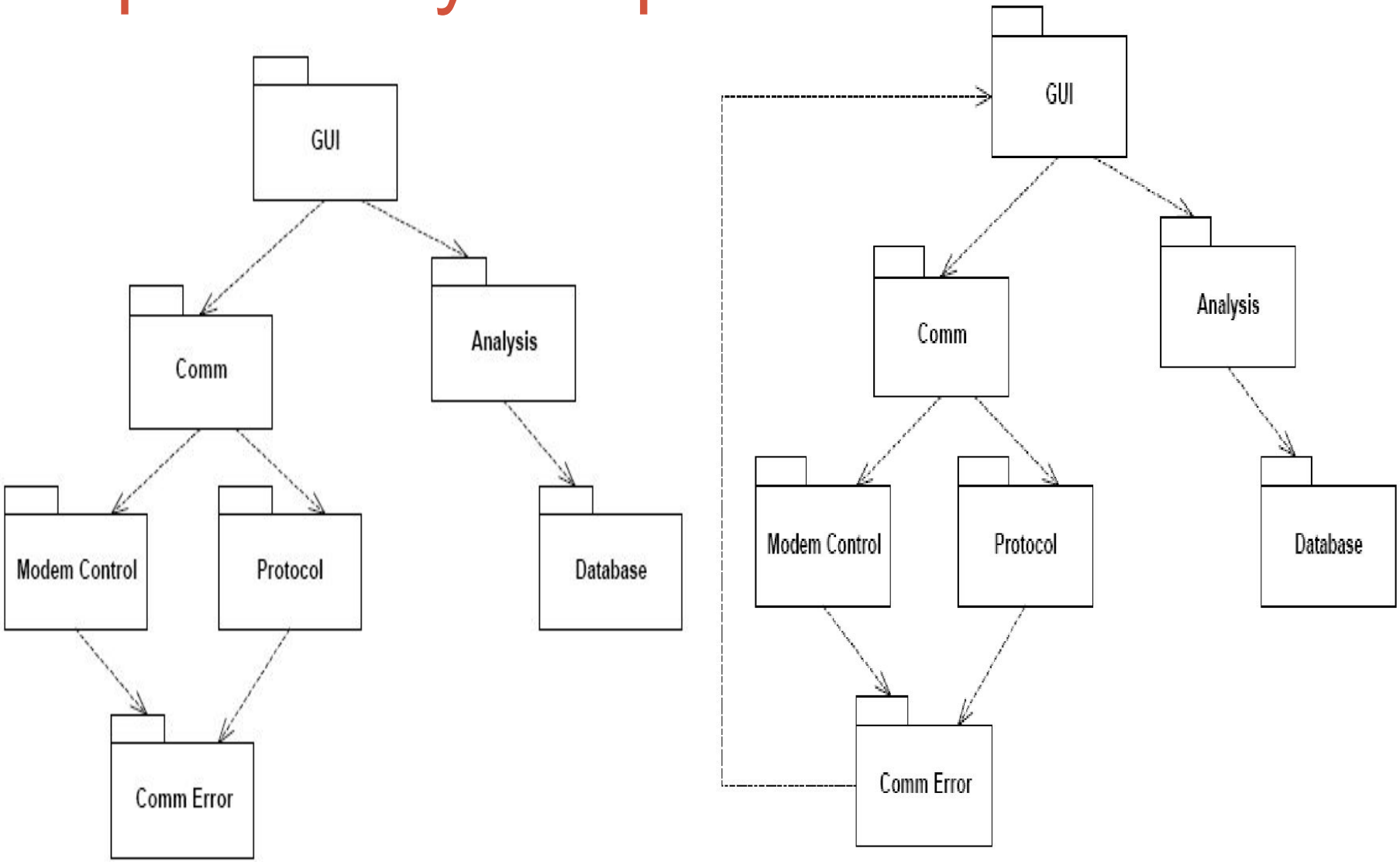
Acyclic Dependencies Principles (ADP)

The dependency structure for released component must be a Directed Acyclic Graph (DAG).There can be no cycles.

[R. Martin]

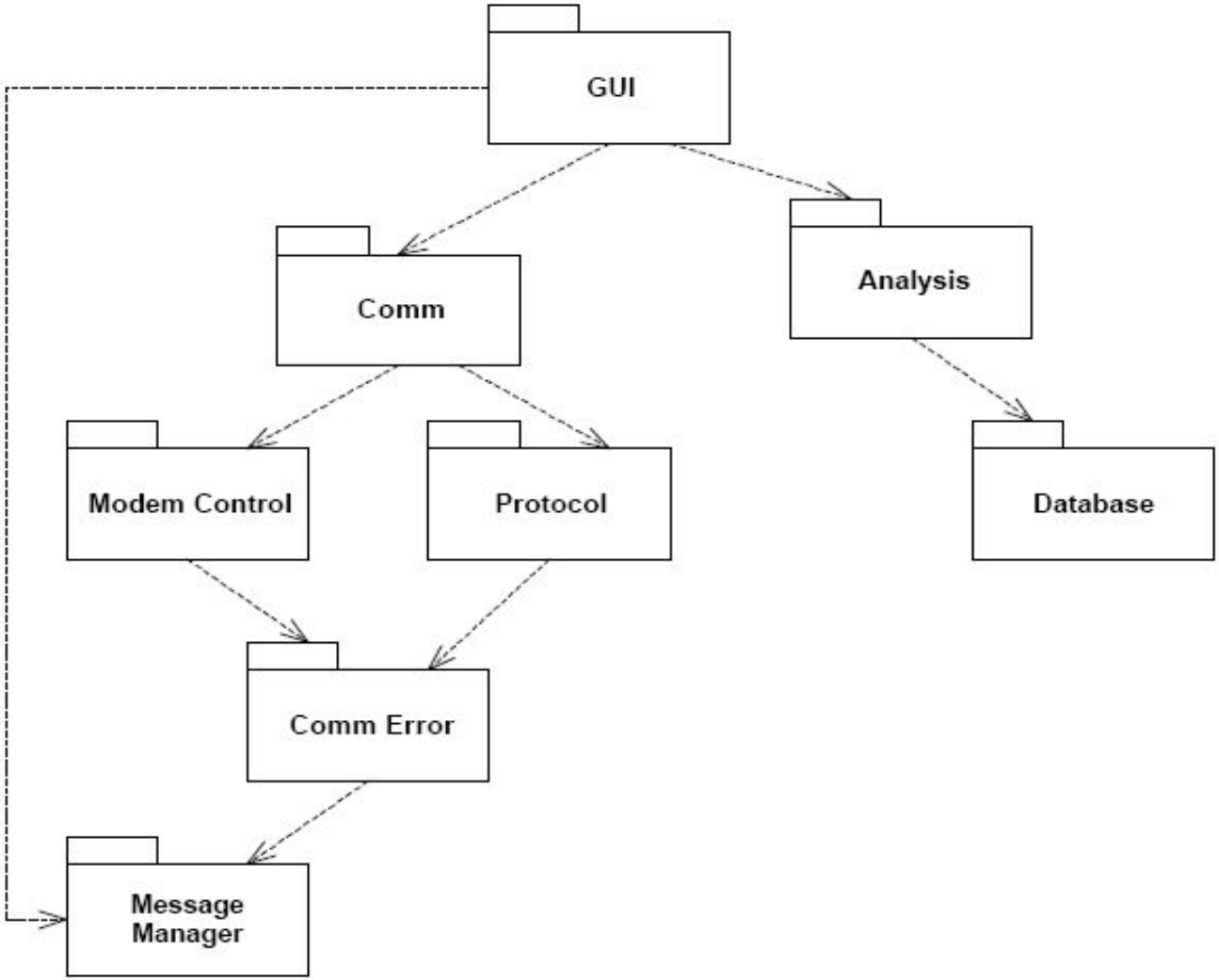


Dependency Graphs



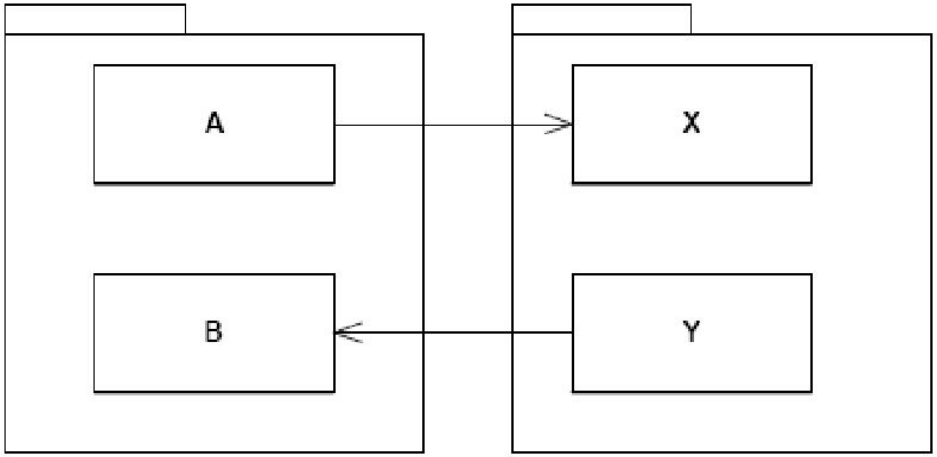
Breaking the Cycle

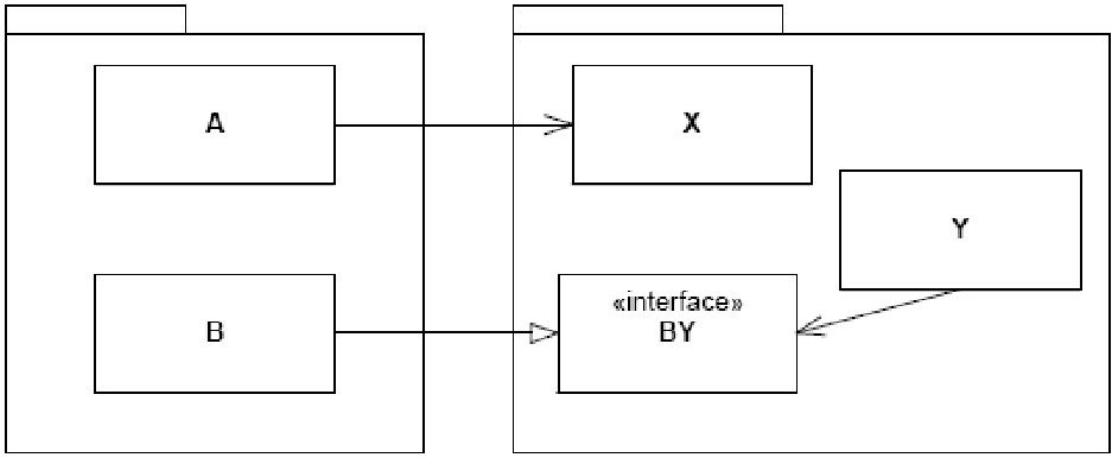
- Add a



Breaking the Cycle

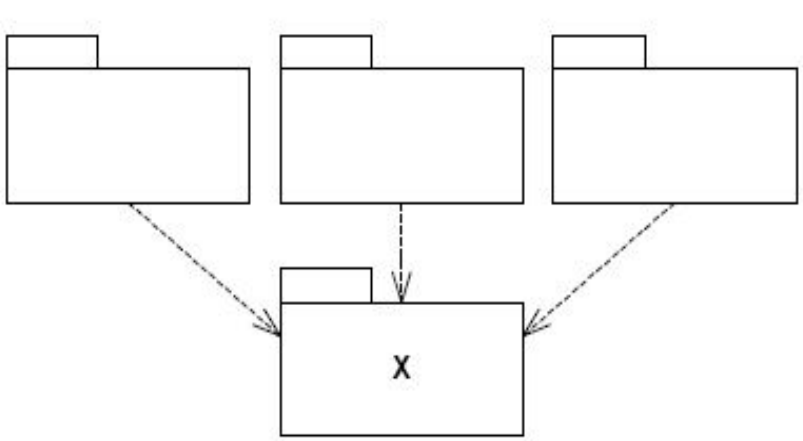
- DIP + ISP

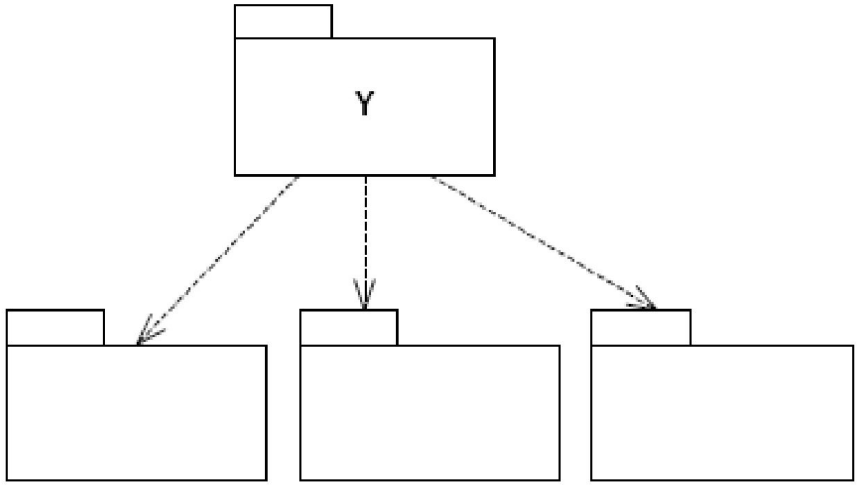




Stability

- Stability is related to the amount of work in order to make a change.





Stability = Responsibility + Independence

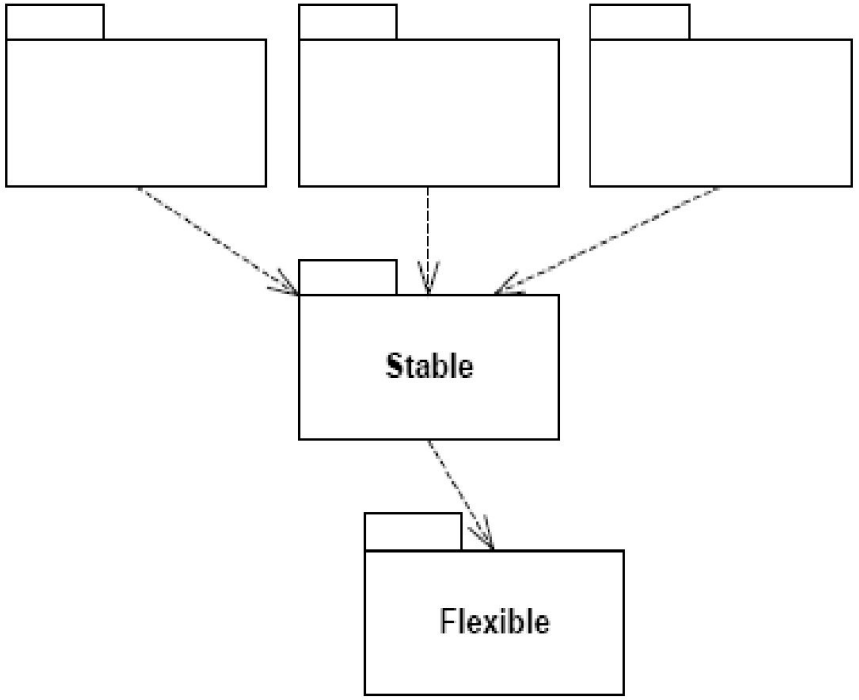
Stability metrics

- *Ca* – Afferent coupling (incoming dependencies)
  - How responsible am I?
- *Ce* – Efferent coupling (outgoing dependencies)
  - How dependant am I?
- *I* = *Ce/(Ca+Ce)* Instability

Example for X:
Ca = 3, *Ce* = 0 => *I* = 0 (very stable)

Stable Dependency Principle (SDP)

- Depend in the direction of stability.
- What does this mean?
  - Depend upon packages whose I is lower than yours.
- Counter-example



Where to Put High-Level Design?

- High-level architecture and design decisions don't change often
  - shouldn't be volatile ⇒ place them in stable packages
  - design becomes hard to change ⇒ *inflexible design*
- How can a totally stable package (I = 0) be flexible enough to withstand change?
  - improve it without modifying it...
- Answer: ***The Open-Closed Principle***
  - classes that can be extended without modifying them ⇒ Abstract Classes

Stable Abstractions Principle (SAP)

- Stable packages should be abstract packages.
- What does this mean?
  - Stable packages should be on the bottom of the design (depended upon)
  - Flexible packages should be on top of the design (dependent)
  - OCP => Stable packages should be highly abstract

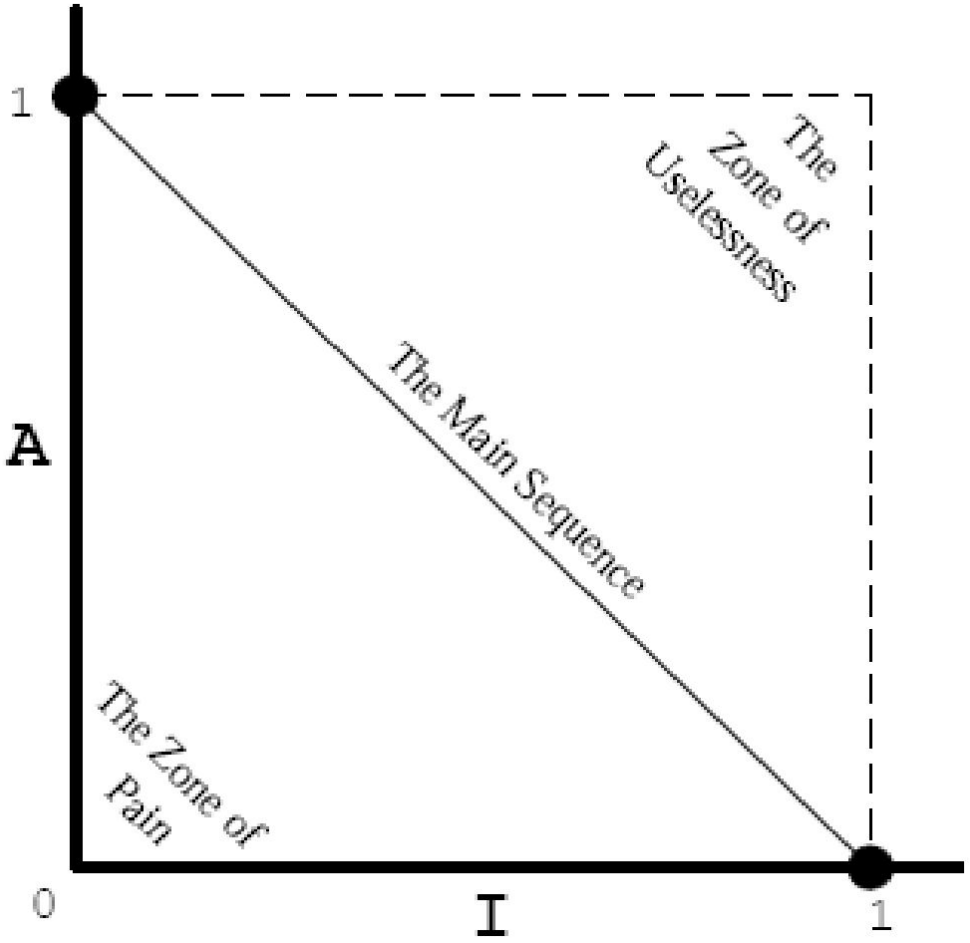
Abstractness metrics

- *Nc* = number of classes in the package
- *Na* = number of abstract classes in the package
- *A* = *Na*/*Nc* (Abstractness)

- Example:
  - *Na* = 0 => A = 0
- What about hybrid classes?

The Main Sequence

- *I* should increase as *A* decreases



The Main Sequence

- Zone of Pain
  - highly stable and concrete ⇒ rigid
  - famous examples:
    - database-schemas (volatile and highly depended-upon)
    - concrete utility libraries (instable but non-volatile)
- Zone of Uselessness
  - instable and abstract ⇒ useless
    - no one depends on those classes
- Main Sequence
  - maximizes the distance between the zones we want to avoid
  - depicts the balance between abstractness and stability.

Why measure?

- "When you can measure what you are speaking about and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind: it may be the beginnings of knowledge but you have scarcely in your thoughts advanced to the stage of Science."
  - Lord Kelvin (Physicist)
- "You cannot control what you cannot measure."
  - Tom DeMarco (Software Engineer)

Why measure?

- Understand issues of software development
- Make decisions on basis of facts rather than opinions
- Predict conditions of future developments

What is Measurement

- measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined, unambiguous rules

Methodological issues

- Measure only for a clearly stated purpose
- Specifically: software measures should be connected with quality and cost
- Assess the validity of measures through controlled, credible experiments
- Apply software measures to software, not people
- Goal-Question-Metric Approach

Examples of Entities and Attributes

- Software Design
  - Defects discovered in design reviews
- Software Design Specification
  - Number of pages
- Software Code
  - Number of lines of code, number of operations
- Software Development Team
  - Team size, average team experience

Types of Metric

- direct measurement
  - eg. number of lines of code
- indirect/ derived measurement
  - eg. defect density = no. of defects in a software product / total size of product
- prediction
  - eg. predict effort required to develop software from measure of the functionality - function point count

Types of metric

- nominal
  - eg no ordering, simply attachment of labels

  (language: 3GL, 4GL)
- ordinal
  - eg ordering, but no quantitative comparison (programmer capability: low, average, high)

Types of metric

- interval
  - eg. between certain values (programmer capability: between 55th and 75th percentile of the population ability)
- ratio
  - eg. the proposed software is twice as big as the software that has just been completed
- absolute
  - eg. the software is 350,000 lines of code long

Types of metric

- product metrics
  - size metrics
  - complexity metrics
  - quality metrics
- process metrics
- resource metrics
- project metrics

Product metric Example 1 - size

- Number of Lines of Code (NLOC)
  - number of delivered source instructions (NDSI)
  - number of thousands of delivered source instructions (KDSI)
- Definition (Conte 1986)

"A line of code is any line of program text that is not a comment or a blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements."

Pros and cons

- Pros as a cost estimate parameter:
  - Appeals to programmers
  - Fairly easy to measure on final product
  - Correlates well with other effort measures
- Cons:
  - Ambiguous (several instructions per line,…)
  - Does not distinguish between programming languages of various abstraction levels
  - Low-level, implementation-oriented
  - Difficult to estimate in advance

Product metric Example 2 - size

- Function Point Count
  - A measure of the functionality perceived by the user delivered by the software developer. A function count is a weighted sum of the number of
    - inputs to the software application
    - outputs from the software application
    - enquiries to the software application
    - data files
      - internal to the software application
      - shared with other software applications

Pros and cons

- Pros as a cost estimate parameter:
  - Relates to functionality, not just implementation
  - Experience of many years, ISO standard
  - Can be estimated from design
  - Correlates well with other effort measures

- Cons:
  - Oriented towards business data processing
  - Fixed weights

Product metric Example - complexity

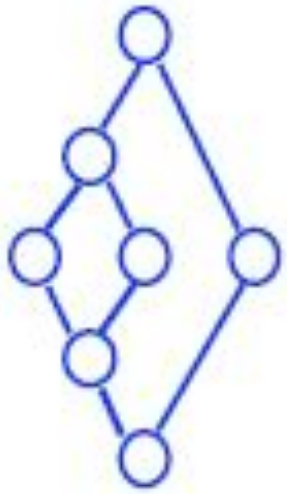
- Graph Theoretic Metric
  - The McCabe Complexity Metric
    - a software module can be described by a control flow graph where
      - each node correspond to a block of sequential code
      - each edge corresponds to a path created by a decision

Product metric Example - complexity

- V(G) = e - n + 2p
  - e = number of edges in the graph
  - n = number of nodes in the graph
  - p = number of connected module components in the graph

e=8
n=7
p=2

V(G)=5



Cyclomatic complexity (CC)

- CC = Number of decisions + 1
- Variants:
  - CC2 Cyclomatic complexity with Booleans ("extended cyclomatic complexity")

    CC2 = CC + Boolean operators

  - CC3 Cyclomatic complexity without Cases ("modified cyclomatic complexity")

    CC3 = CC where each Select block counts as one

OO metrics

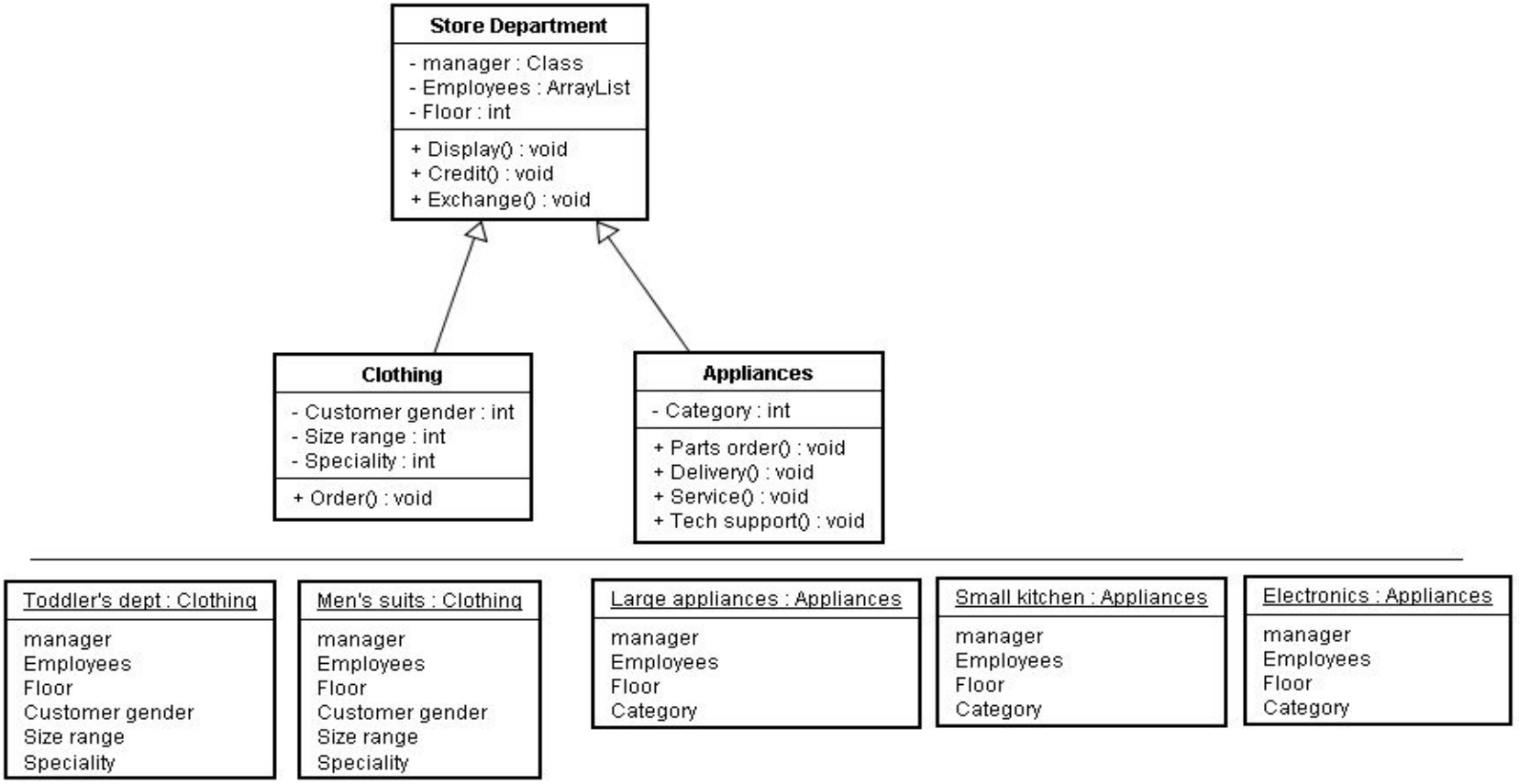
- Weighted Methods Per Class (WMC)
- Depth of Inheritance Tree of a Class (DIT)
- Number of Children (NOC)
- Coupling Between Objects (CBO)
- Response for a Class (RFC)
- Lack of Cohesion (LCOM)

Weighted Methods Per Class (WMC)

- Sum of the complexity of each method contained in the class.
- Method complexity: (e.g. cyclomatic complexity)
  - When method complexity assumed to be 1, WMC = number of methods in class

Example

- WMC for *Clothing* = 1
- WMC for *Appliance* = 4



Depth of Inheritance Tree of a Class (DIT)

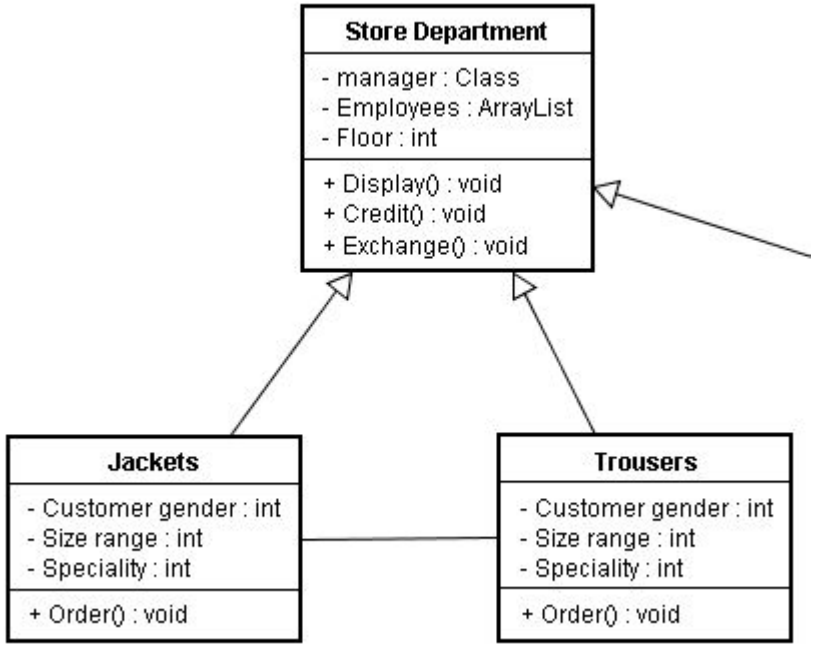
- is the maximum number of steps from the class node to the root of the tree and is measured by the number of ancestor classes
- DIT *(Store_Dept)* = 0.
- DIT (*Clothing)* = 1

Number of children (NOC)

- Number of immediate subclasses of a class.
- NOC(*Store_Dept*) = 2
- NOC(*Clothing*) = 0

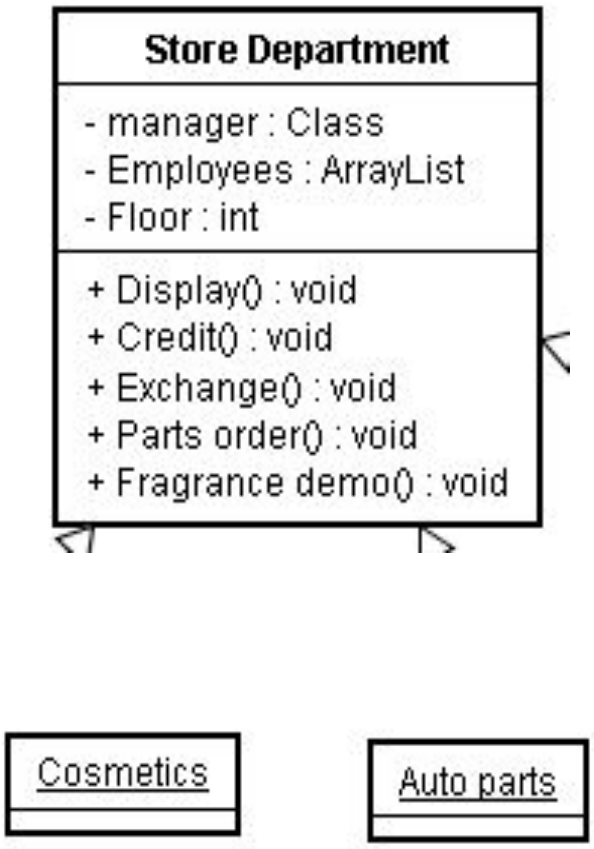
Coupling between objects (CBO)

- Number of other classes to which a class is coupled, i.e., suppliers of a class.
- Two classes are coupled when methods declared in one class use methods or instance variables defined by the other class.
- The uses relationship can go either way: both uses and used-by relationships are taken into account, but only once.



Lack of cohesion (LCOM)

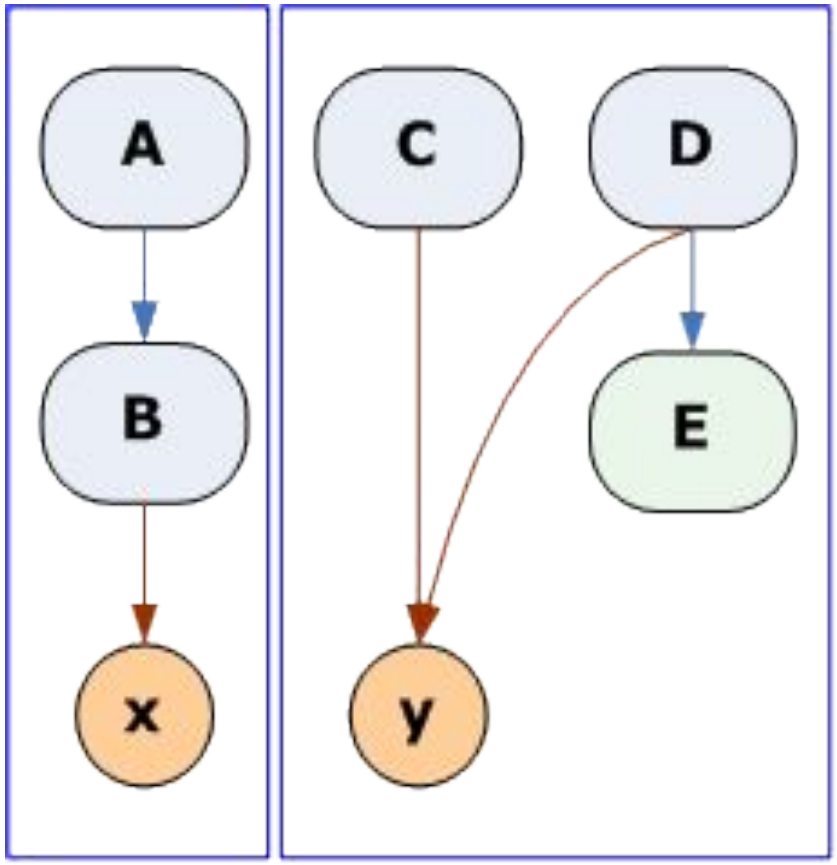
- LCOM measures the dissimilarity of methods in a class by instance variable or attributes.
- Several variants
- LCOM4 recommended



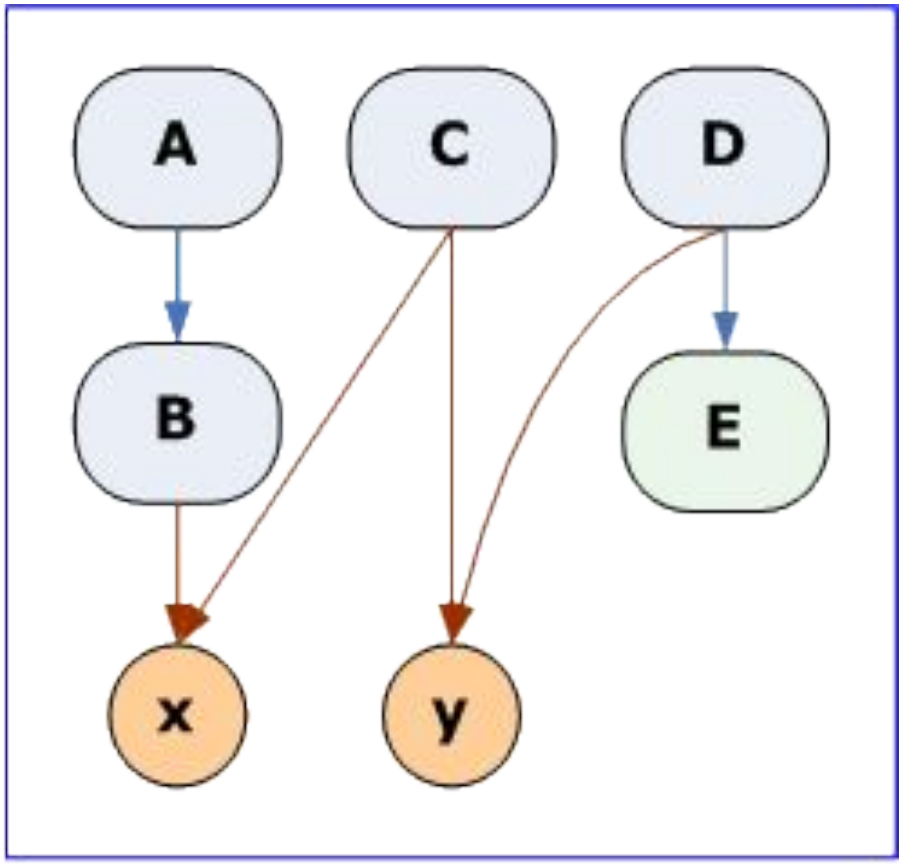
LCOM4

- LCOM4 measures the number of *"connected components"* in a class.
- A connected component is a set of related methods (and class-level variables). There should be only one such a component in each class. If there are 2 or more components, the class should be split into so many smaller classes.
- Which methods are related? Methods a and b are related if:
  - they both access the same class-level variable, or
  - a calls b, or b calls a.

LCOM4



LCOM4 = 2



LCOM4 = 1

Response for a Class (RFC)

- The RFC is the count of the set of all methods that can be invoked in response to a message to an object of the class or by some method in the class. This includes all methods accessible within the class hierarchy.

- RFC (*Store_dept)* = 3 (self) + 1 (*Clothing*) + 4 (*Appliance*) = 8

Summary

Metrics	Objective	Testing Efforts	Understan -dability	Maintain -ability	Develop Effort	Reuse
Complexity	↓	↓	↑	↑		
Size (LOC)	↓	↓	↑	↑		
Comment %	↑	↓	↑	↑	↓	
WMC	↓			↑	↓	↑
RFC	↓	↓				↑
LCOM	↓		↑	↑	↓	↑
CBO	↓	↓	↑	↑		↑