Ulteriori principi

• The Interface Segregation Principle: Many client specific interfaces are better than one general purpose interface

• The Reuse/Release Equivalency Principle: The granule of reuse is the same as the granule of release. Only components that are released through a tracking system can be effectively reused

• The Acyclic Dependencies Principle: The dependency structure for released components must be a directed acyclic graph. There can be no cycles

• The Stable Dependencies Principle: A package should only depend upon packages that are more stable than it is

• The Stable Abstractions Principle: Packages that are maximally stable should be maximally abstract. Unstable packages should be concrete. The abstractness of a package should be in proportion to its stability
Metrics

Taken from material by
Michael L. Collard
Definitions

• Measure - quantitative indication of extent, amount, dimension, capacity, or size of some attribute of a product or process.
  – E.g., Number of errors

• Metric - quantitative measure of degree to which a system, component or process possesses a given attribute. “A handle or guess about a given attribute.”
  – E.g., Number of errors found per person hours expended
Why Measure Software?

- Determine the quality of the current product or process
- Predict qualities of a product/process
- Improve quality of a product/process
Examples

- Estimate the cost & schedule of future projects
- Evaluate the productivity impacts of new tools and techniques
- Establish productivity trends over time
- Improve software quality
- Forecast future staffing needs
- Anticipate and reduce future maintenance needs
Metric Classification

• Products
  – Explicit results of software development activities
  – Deliverables, documentation, by products

• Processes
  – Activities related to production of software

• Resources
  – Inputs into the software development activities
  – hardware, knowledge, people
Product vs. Process

• Process Metrics
  – Insights of process paradigm, software engineering tasks, work product, or milestones
  – Lead to long term process improvement

• Product Metrics
  – Assesses the state of the project
  – Track potential risks
  – Uncover problem areas
  – Adjust workflow or tasks
  – Evaluate teams ability to control quality
Types of Measures

• Direct Measures (internal attributes)
  – Cost, effort, LOC, speed, memory

• Indirect Measures (external attributes)
  – Functionality, quality, complexity, efficiency, reliability, maintainability
Size-Oriented Metrics

- Size of the software produced
- LOC - Lines Of Code
- KLOC - 1000 Lines Of Code
- SLOC – Statement Lines of Code (ignore whitespace)
- Typical Measures:
  - Errors/KLOC, Defects/KLOC, Cost/LOC, Documentation Pages/KLOC
McCabe’s Complexity Measures

• McCabe’s metrics are based on a control flow representation of the program
• A program graph is used to depict control flow
  – Nodes represent processing tasks (one or more code statements)
  – Edges represent control flow between nodes
• Cyclomatic Complexity: set of independent paths through the graph (basis set)
Measures of Software Quality

- **Correctness** – degree to which a program operates according to specification
  - Defects/KLOC
  - Defect is a verified lack of conformance to requirements
  - Failures/hours of operation
- **Maintainability** – degree to which a program is open to change
  - Mean time to change
  - Change request to new version (Analyze, design etc)
  - Cost to correct
- **Integrity** – degree to which a program is resistant to outside attack
  - Fault tolerance, security & threats
- **Usability** – easiness to use
  - Training time, skill level necessary to use, Increase in productivity, subjective questionnaire or controlled experiment
Chidamber and Kemerer Metrics

- Weighted methods per class (MWC)
- Depth of inheritance tree (DIT)
- Number of children (NOC)
- Coupling between object classes (CBO)
- Response for class (RFC)
- Lack of cohesion metric (LCOM)
WMC: Weighted Methods per Class

• WMC = \( \sum_{i=1}^{n} c_i \)

• \( c_i \) is the complexity (e.g., volume, cyclomatic complexity, etc.) of each method

• The number of methods and complexity of methods is an indicator of how much time and effort is required to develop and maintain the object

• The larger the number of methods in an object, the greater the potential impact on the children

• Objects with large number of methods are likely to be more application specific, limiting the possible reuse
DIT: Depth of Inheritance Tree

- Maximum length from a node to the root (base class)
- Lower level subclasses inherit a number of methods making behavior harder to predict
- Deeper trees indicate greater design complexity
NOC: Number of Children

- Number of subclasses immediately subordinate to a class
- As NOC grows, reuse increases - but the abstraction may be diluted
- Depth is generally better than breadth in class hierarchy, since it promotes reuse of methods through inheritance
- Classes higher up in the hierarchy should have more subclasses than those lower down
- NOC gives an idea of the potential influence a class has on the design: classes with large number of children may require more testing
CBO: Coupling between Objects/Classes

• Number of collaborations between two classes (fan-out of a class C)
  – the number of other classes that are referenced in the class C
  – A reference to another class, A, is a reference to a method or a data member of class A

• As collaboration increases reuse decreases
• High fan-outs represent class coupling to other classes/objects and thus are undesirable
• High fan-ins represent good object designs and high level of reuse
• Not possible to maintain high fan-in and low fan outs across the entire system
RFC: Response for a Class

• Number of methods that could be called in response to a message to a class (local + remote)

• As RFC increases
  – testing effort increases
  – greater the complexity of the object
  – harder it is to understand
LCOM: Lack of Cohesion in Methods

• Number of pairs of methods in a class that don't have at least one field in common minus the number of pairs of methods in the class that do share at least one field.
• When this value is negative, the metric value is set to 0.
• Lower metric values represent 'better' situations.
Class Size

• Number of operations
  – inherited, private, public
• Number of attributes
  – inherited, private, public

• May be an indication of too much responsibility for a class
NOO: Number of Operations Overridden

- A large number for NOO indicates possible problems with the design
- Poor abstraction in inheritance hierarchy
NOA: Number of Operations Added

• Number of operations added by a subclass

• As operations are added it is farther away from super class

• As depth increases NOA should decrease
Metriche di stabilità

Lucidi tratti da materiale di Angelo Gargantini
### Metriche di stabilità

- **CC**: numero di classi concrete in un package P
- **AC**: numero di classi astratte o di interfacce in un package P
- **Ca** (afferent coupling): numero di package che dipendono da classi del package P
- **Ce** (efferent coupling): numero di package da cui dipendono le classi del package P
- **A** (Astrattezza): percentuale di classi astratte del package P, definita come il rapporto AC / (AC + CC)
- **I** (Instabilità): rapporto fra l'accoppiamento efferente e l'accoppiamento totale Ce / (Ce + Ca)
  - È un indicatore della difficoltà di modificare il package P (se infatti è alto, ciò significa che molte classi dipendono da esso).
- **D** (Distanza dalla sequenza principale): distanza del package P dalla retta di equazione \( A + I = 1 \). E' un indicatore del compromesso raggiunto dal package fra astrattezza \( (A = 1, I = 0) \) e instabilità \( (A = 0, I = 1) \)
Diversi casi
Cosa fare

• Spostare delle classi concrete in package instabili
• Rendere astratte classi in packages stabili
Dipendenze tra package

- Un package A che importa da un altro B e B importa da A
- Oppure indirette
• Rompere i cicli è molto difficile
• Vedere dove è il ciclo e spostare metodi/classi
• Rivela in genere un problema di progettazione
FindBugs

• Based on the concept of bug patterns. A bug pattern is a code idiom that is often an error.
  – Difficult language features
  – Misunderstood API methods
  – Misunderstood invariants when code is modified during maintenance
  – Garden variety mistakes: typos, use of the wrong boolean operator
• FindBugs uses static analysis to inspect Java bytecode for occurrences of bug patterns.
• Static analysis means that FindBugs can find bugs by simply inspecting a program's code: executing the program is not necessary.
• FindBugs works by analyzing Java bytecode (compiled class files), so you don't even need the program's source code to use it.
• FindBugs can report false warnings, not indicate real errors. In practice, the rate of false warnings reported by FindBugs is less than 50%.
PMD

• PMD scans Java source code and looks for potential problems like:
• Possible bugs - empty try/catch/finally/switch statements
• Dead code - unused local variables, parameters and private methods
• Suboptimal code - wasteful String/StringBuffer usage
• Overcomplicated expressions - unnecessary if statements, for loops that could be while loops
• Duplicate code - copied/pasted code means copied/pasted bugs
PMD Basic Rules

- **EmptyCatchBlock**: Empty Catch Block finds instances where an exception is caught, but nothing is done. In most circumstances, this swallows an exception which should either be acted on or reported.
- **EmptyIfStmt**: Empty If Statement finds instances where a condition is checked but nothing is done about it.
- **EmptyWhileStmt**: Empty While Statement finds all instances where a while statement does nothing. If it is a timing loop, then you should use `Thread.sleep()` for it; if it's a while loop that does a lot in the exit expression, rewrite it to make it clearer.
- **EmptyTryBlock**: Avoid empty try blocks - what's the point?
- **EmptyFinallyBlock**: Avoid empty finally blocks - these can be deleted.
- **EmptySwitchStatements**: Avoid empty switch statements.
- **JumbledIncrementer**: Avoid jumbled loop incrementers - it's usually a mistake, and it's confusing even if it's what's intended.
- **ForLoopShouldBeWhileLoop**: Some for loops can be simplified to while loops - this makes them more concise.
Sonar

- Dashboard to summarize Static and Dynamic analysis Tools
- Conventions (Checkstyle)
- Bad Practices (PMD)
- Potential Bugs (FindBugs)