



# Model Static Analysis



## Introduction

Traditional **Static Analysis** is a well-known discipline in the world of software development. While dynamic analysis during the system execution performs the working system black-box testing, the static analysis focuses on the source code, its internal structure and organization, relationships between its modules, architecture, etc. There are a lot of software tools for the source code analysis for the practically every programming language and data base available.

The most of the software complexity is hidden in its static structure, determined by **software elements** (classes, database tables, web pages, stored procedures, etc) and **dependencies between them** (relationships, method invocations, etc). Static analysis is, though, a valuable tool for this complexity management and control.

*Software structure management and control is a requisite for the robust, extensible and easy to understand system construction.*



**Enterprise Analyst** incorporates a powerful tool for the UML class model static analysis. Its design is based on and guided by the traditional static analysis methods, raised to the higher abstraction level. Thanks to Enterprise Analyst, the software teams can start with the complexity control of their systems from the model level!

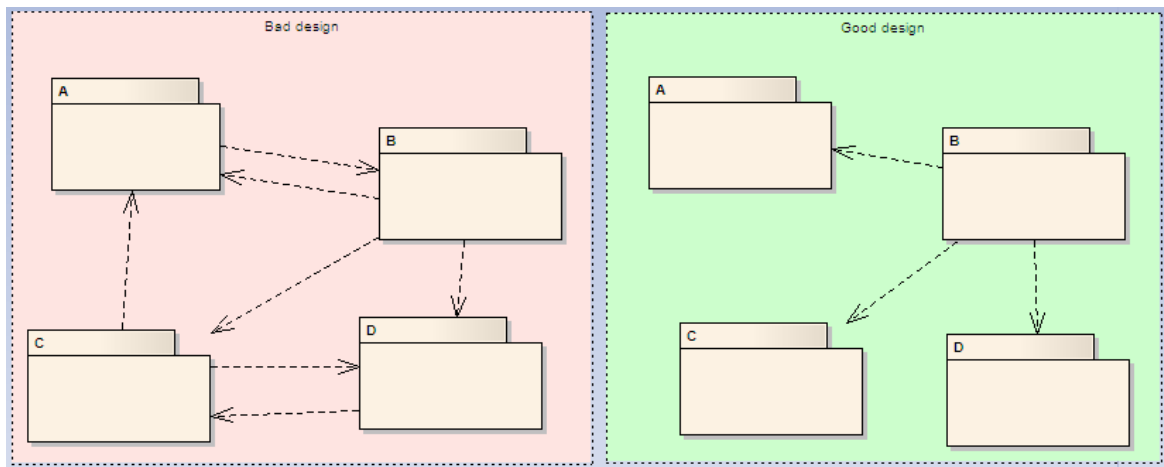
In this document the theoretical knowledge base of the model static analysis is exposed. It is divided in three sections:

- Model dependencies basics
- Dependency matrix, and
- Model metrics

## Model Dependencies

Internal dependencies between its modules and elements originate the complexity of a software system (or model). If there are lot of connections, references, and dependencies, any modification anywhere in the system might cause crashes anywhere else, or in the best case - the need to make a lot of updates on other elements.

Without a need for an explicit and formal analysis, it is easy to have an idea about the potential quality of the following two structures, composed of only 4 modules:



*Bad design versus Good design*

The "Bad Design" solution: This design is very "tangled", it has too many dependencies between its modules, including the most dangerous ones - the cyclic dependencies (so called **tangles**). A structure like this is not very resistant to modifications and updates, since they propagate easily through the dependency networks.

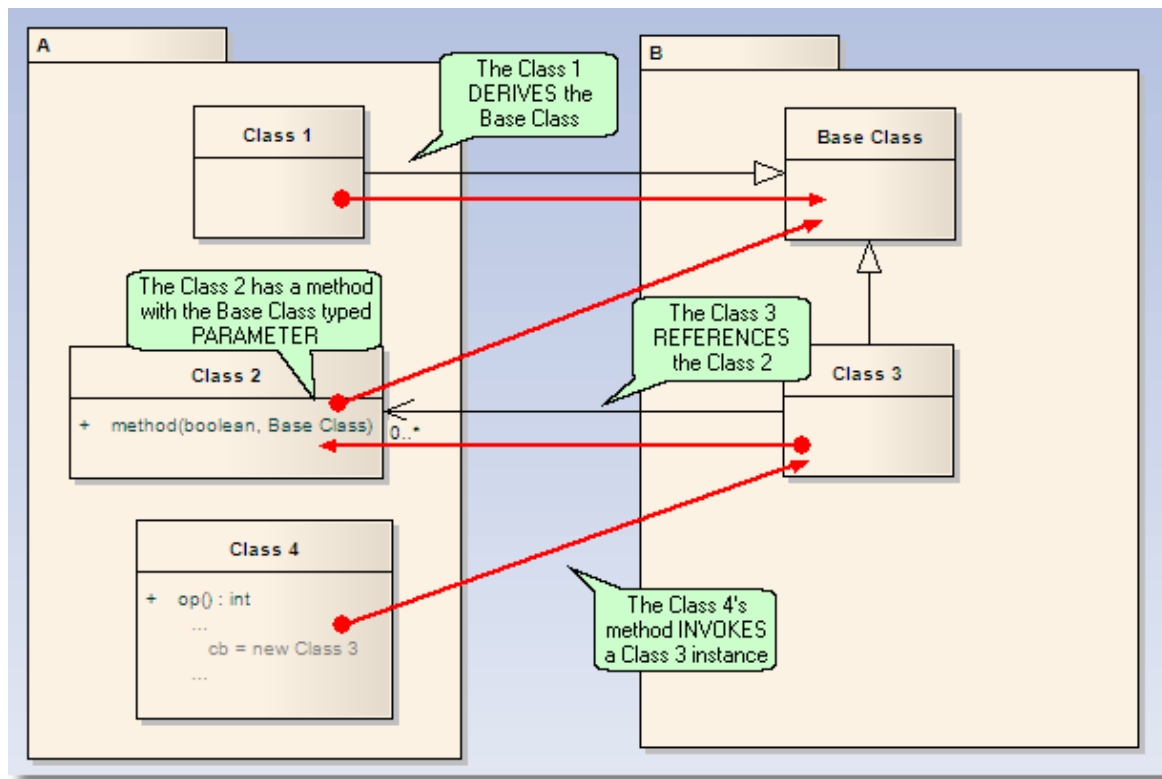
For example, a change made to module D might propagate directly to modules B or C, and indirectly to A. Additionally, those updates might even "rebound" and get back to the module D, following the cyclic dependencies, and possibly making D change again. Actually, in this design any modification propagates aggressively to the rest of the system.

The "Good Design" solution: This alternative is without any doubt much better. Reasonably low dependency count with no cycles (tangles) makes it easy to control and extend this model.

A modification in module D can now eventually propagate to B. Same happens with a modification in modules A or C. Module B is completely free of incoming dependencies, so it can be freely modified without any risk of change propagation or impact on the rest of the system!


### Dependency Sources

Dependencies between the modules (or packages) are actually "derived" dependencies. "Real" ones are between the low-level elements (classes, tables, procedures, etc) contained in the corresponding packages. The following diagram explains some different dependency sources between the model elements (in this case, classes). The red arrows indicate the dependency direction (from the depending class to the class on which it depends):



Sources of dependencies in a model

As a consequence of those dependencies, the package A depends on the package B (Class 1 and Class 2 depend on Base Class and Class 4 depends on the Class 3), and vice versa (Class 3 depends on Class 2).

 **Enterprise Analyst** detects and analyses all 4 possible dependency sources: inheritance, reference, method parameter and invocation.

## Dependency weight

As a consequence of dependencies shown on the last figure, the package A depends on the package B, as well as the package B on the package A. An inside look at those dependencies' sources discovers the that actually there are 3 dependencies from A to B and only one from B to A. Package A depends on 2 classes from the package B, while package B depends on only one class from the package A. The dependency from A to B is "**heavier**" than the B to A dependency! This dependency is **harder** to manage.

This phenomenon is known as **dependency weight**.

There are 4 different strategies to calculate dependency weight:

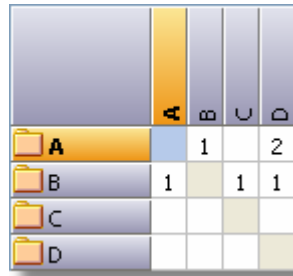
- *Binary strategy* - the logical value expressing whether there IS or there IS NOT dependency between two packages
- *Knowledge-based subsystem strategy* - represents the number of classes from a package, known to the classes from depending one
- *Knowledge-based class strategy* - represents the number of classes from depending package that know about the classes in another one
- *Usage strategy* - calculates the total of atomic dependencies from one package to another



The current version of **Enterprise Analyst** applies the Usage Strategy.  
The future versions will gradually incorporate remaining strategies.

## Dependency Matrix

A model can have hundreds or even thousands of classes and packages. The dependencies count is even higher. To simplify the dependency structure representation, it is common to use so called **Dependency Matrix (DM)**.



	A	B	C	D
A		1		2
B			1	1
C				
D				

*Dependency Matrix (example 1)*

Rows and columns in a DM correspond to packages and/or classes in the model. The numbers in its cells represent the dependency weights that correspond to a dependency between the column-element and row-element.

The DM from in the example 1 shows the system composed from 4 modules (packages): A, B, C and D. The package D depends on the packages A (weight 2) and B (weight 1). The package C depends only on the package B (weight 1).

The MD has some algebraic characteristics, which enable rapid architectural diagnostics before diving into the profound analysis using the architectural metrics.

- the main matrix diagonal cells are always empty, since an element does not depend on itself
- for a model free of tangles (dependency cycles) it is always possible to distribute the non-empty cells on just one side of the main diagonal
- tangles oblige to have non-empty cells on the both sides of the main diagonal
- non-empty cells in a single row identify the elements that depend on the element associated to that row
- non-empty cells in a single column identify the elements which the element associated to the column depend on
- a lot of non-empty cells is an indicator of a highly coupled subsystems (packages)

From the example 1 it is easy to detect the tangle formed of the packages A and B. All the cells from the row B are non-empty, which means that the rest of the subsystems depend on B. The column D discovers that this package depends on A and B, etc.

The example 2 shows a more detailed view into the same system. In this figure, the Matrix shows the dependencies between the classes that "live" in the subsystems A, B, C and D (from the Example 1):

	A1	A2	A3	B1	B2	C1	C2	D1	D2	D3	D4
A1					1			1		1	
A2			1								
A3											
B1	1										
B2				1			1		1		
C1											
C2											
D1											
D2											1
D3											1
D4									1		

*Example 2: The Class Dependency Matrix*

The dependency between the packages C and B (detected in the previous example) is "unmasked" in this example. The Class C2 (from the package C) depends on the B2 (from B).



**Enterprise Analyst** offers a full support of the both styles of Dependency Matrix.

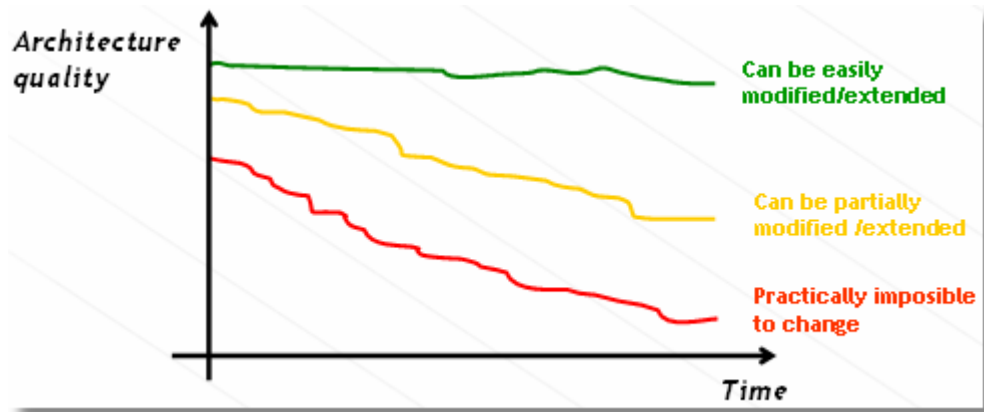
Using the model DM it is relatively easy to quickly analyze the overall dependency structure, detect the tangles and begin the architectural diagnostics.

For detailed analysis, the model metrics are used.

## Model Metrics

The model architecture metrics define a set of numeric model attributes, expressing objectively some of its important properties. Models, just like systems, tend to degrade in time. Each model update or extension, affects its properties, typically eroding its quality. Lower is the model architecture quality, harder is to make further changes.

**Permanent control over the model metrics permits to slow down the model degradation process**



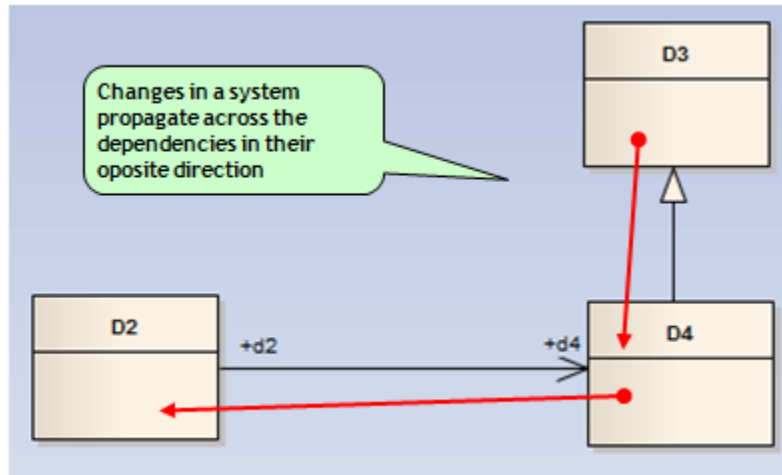
*Model architecture degradation*

Just like this figure shows, certain model properties can be quantified and tracked through time. This way, model evolution can be monitored and controlled.

In this section, the architecture metrics are described. All of them apply to a single subsystem or a model as a whole.

### Average Impact (AI)

AI is a measure of potential damage done to a model when some part of is changed/modified. It is calculated as arithmetic average of the Impacts produced by changing its classes. A single class impact is abstracted as a number of classes which depend directly or indirectly on this class. For example, the impact of a class D3 on the following diagram is 2 (D4 depends on D3 and D2 depends on D4):



*Impact and system stability*

Likewise, the impact of the class D4 is 1, and the D2's is 0 (no classes depend on D2).

$$AI = \frac{\sum I}{N}$$

The Average Impact of this subsystem AI = 1.

The AI value is a decimal number from the range [0..N], where N is a total of classes in a model.

**System Stability (SS)**

SS is a normalized AI and is typically expressed in percents.

$$SS = 1 - \frac{AI}{N}$$

Closer the SS is to 100%, less damage is potentially done by changing some of its parts, that is, the model is **more stable**.

**Outgoing and Ingoing Dependencies (OD and ID)**

OD and ID represent the number of dependencies that go "out from" or "in to" particular subsystem. This metric value is a non negative integer.

These metrics are used to calculate other metrics.

**Class Count (C)**

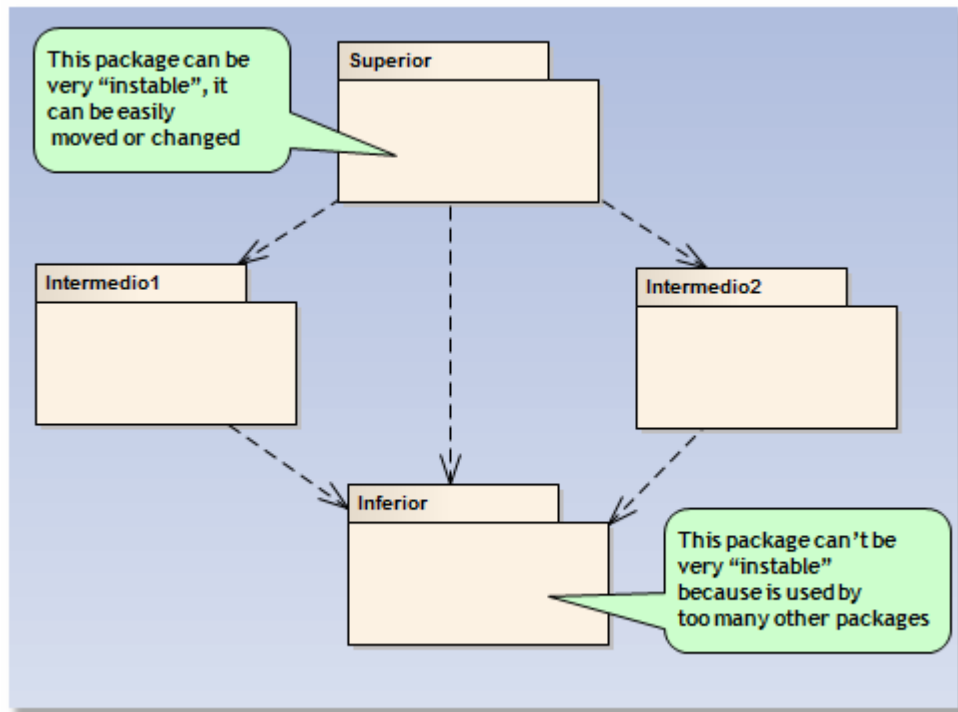
This metric simply shows a total of classes in a subsystem or model. It is used to calculate some other values, averages, etc.

**Abstract Class Count (AC)**

The number of abstract classes in a subsystem/model.

**Instability (I)**

This is a function of a subsystem's outgoing and ingoing dependencies. The idea behind this metric is the following: if a subsystem depends on the others more that the others depend on it, it is easier to change it. If a subsystem has a lot of other subsystems depending on it, it is more dangerous to change it.



*Inestabilidad*

The Instability formula:

$$I = \frac{OD}{OD+ID}$$

The Instability ranges between [0..1].

### Abstractness (A)

Abstractness is a measure of how abstract or concrete is a subsystem. Abstract classes make subsystem more abstract. The metric is calculated as a relation between the number of abstract classes and the total of classes.

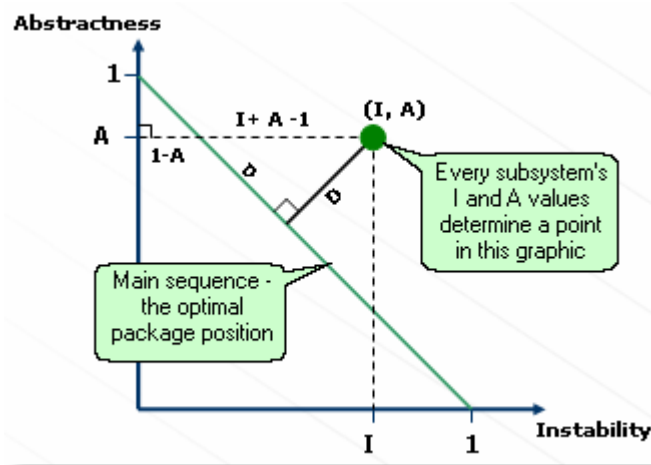
$$A = \frac{AC}{C}$$

This number also takes values between [0..1].

### Distance (D)

Abstractness and Instability are closely related metrics. An instable subsystem should be less abstract and vice versa. This stands for the following reason: a stable system should not change too much, so it is made more abstract (less possibility to change). Other subsystems typically depend on abstract-stable subsystems. On the other side, a subsystem intended to change (instable) should be more concrete. It depends on other subsystems.

The following Figure show the Abstractness-Instability relationship graphically.



*Distance (Abstractness versus Instability)*

Well architected subsystems tend to be localizes around the Main Sequence - the line between the abstract-stable and concrete-instable extremes.

This metric normalized value is:

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

The ideal value for any subsystem is 0 or as close to 0 as possible.

**Relational Cohesion (RC)**

RC is a measure of a strength of coupling between the classes in a subsystem.

$$RC = \frac{D}{C}$$

where D is a dependency count and C is a class count for the subsystem.

A well designed system should be composed of the low-coupled subsystems with the high relational cohesion.



**Enterprise Analyst** supports all 10 architecture metrics. System Stability, Average Impact, Instability, Abstractness, Distance, Cohesion, Ingoing and Outgoing Dependences, class and abstract class count.