THE COLLECTIVE — RESISTANCE IS FUTILE

by Kris Oosting

Over the past few years, we have performed many reviews on various projects. We frequently observed that there was some level of misunderstanding about how to treat collection classes in Object Interaction Graphs (OIGs).

Also, at least in larger projects, there wasn't always a clear view on when a class was reused in design and how this could be measured easily.

This article describes some extensions to Fusion for modeling collections in OIGs. We introduced these to overcome the modeling issues around collections and reuse.

Collections Scenario

Reviewer: Why do you have an object with the same name in your diagram? One is a collection (dashed line) and the other an object (solid line)?

Developer: Well, I have to know when a message goes to the collection (solid line) and when the message has to go to the members of the collection.

Actually, the question of how to model this form of messaging often arises even before the reviews.

Another common burning question is: "How do I know the class of the collection itself?"

The Collection Adaptation

We wanted to adapt Fusion in a way that is clear and simple. We did not like developers having to draw two objects to model messaging to collections as explained by the developer above. The following had to be annotated:

- messaging to the collection itself
- messaging to the members of the collection
- a way to explain what the class of the collection is

To keep the change to a minimum, we decided that when a message is sent to the collection object itself, there is no [select predicate; stop predicate] under the message arrow in the OIG. If a message has to be sent to objects in the collection we use [select predicate; stop predicate].

For example: a message that has to be sent to all objects in the collection has [all] as predicate under the arrow. A message to a selection of objects still uses the standard [select predicate; stop predicate].

Figure 1 shows the OIG notation.

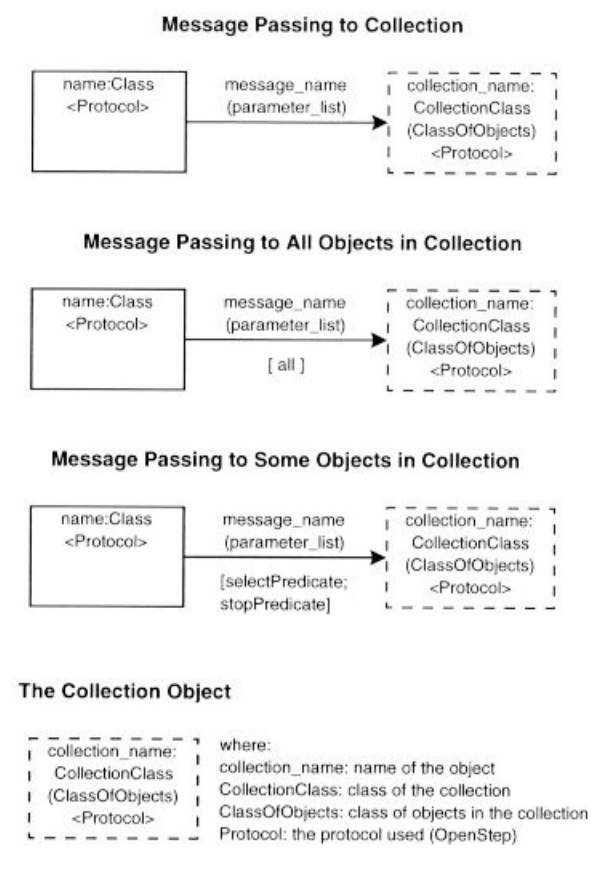


Figure 1 The adapted OIG notation

Our approach to where to put the class of the collection itself is quite simple. We were used to explaining collections in our courses as follows: "the collection object **files** contains a **list** of objects of Class **File**." See Figure 2 for the resulting OIG. The example is taken from page 65-67 of the first Fusion book (Coleman et al., 1994). Figure 3 shows the corresponding Visibility Graph notation.

Reuse Scenario

Reviewer: "Where is the description of this method?"

Developer: "I don't have to do that! Martin is mainly responsible for this class! I'm just using it here."

Reviewer: "Sorry"

Other questions we received were along the lines of: "How do I model my using his message or adding my own to this object? We're working both on this one."

In larger projects it sometimes (actually more than sometimes) happens that two or more developers are working on the same class. We will not enter the discussion of whether this is good or bad—it's just a fact Jack. And we

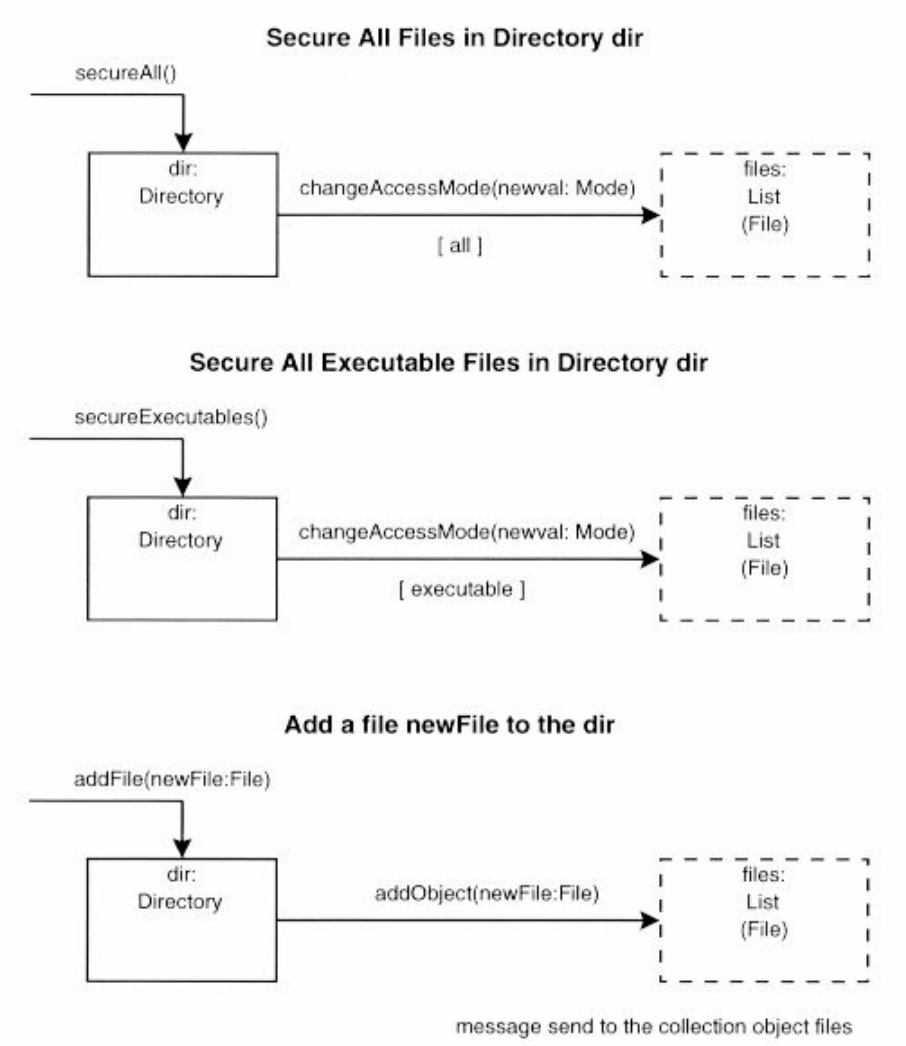


Figure 2 An example using the modified OIG notation

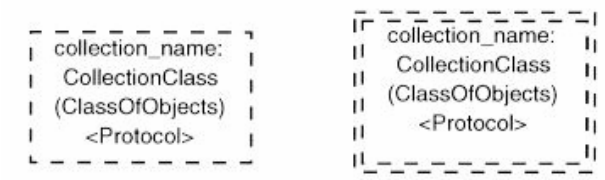


Figure 3 The Visibility Graph Notation

have to deal wi th it. Real life is not always as clean as we would like it.

Ok, back to the class.

The Reuse Adaptation

We had already made adaptations to indicate if a class was reused or not in OIGs and Visibility Graphs (VGs). This is the **lib** prefix for the name o f the object. We did th is, because we wanted to know quickly what the percentage of reuse is in design.

Using the **lib** prefix for solving the other issues about working together on a class was nice, becau se the developers already knew it.

The following was the result of a short discussion:

• If a class (object) is completely reused then it has the **lib** prefix and the messages to this object do not have a prefix and no method description in the OIG.

• If a class (object) is developed by two or more developers then

- if a method developed by another is used then it has the **lib** prefix
- if a method to be developed by yourself is used then it has no prefix

Note: classes are reused. You are, o f course, still free to choose a name for an object of that class.

Figure 4 shows the notation for this adaptation.

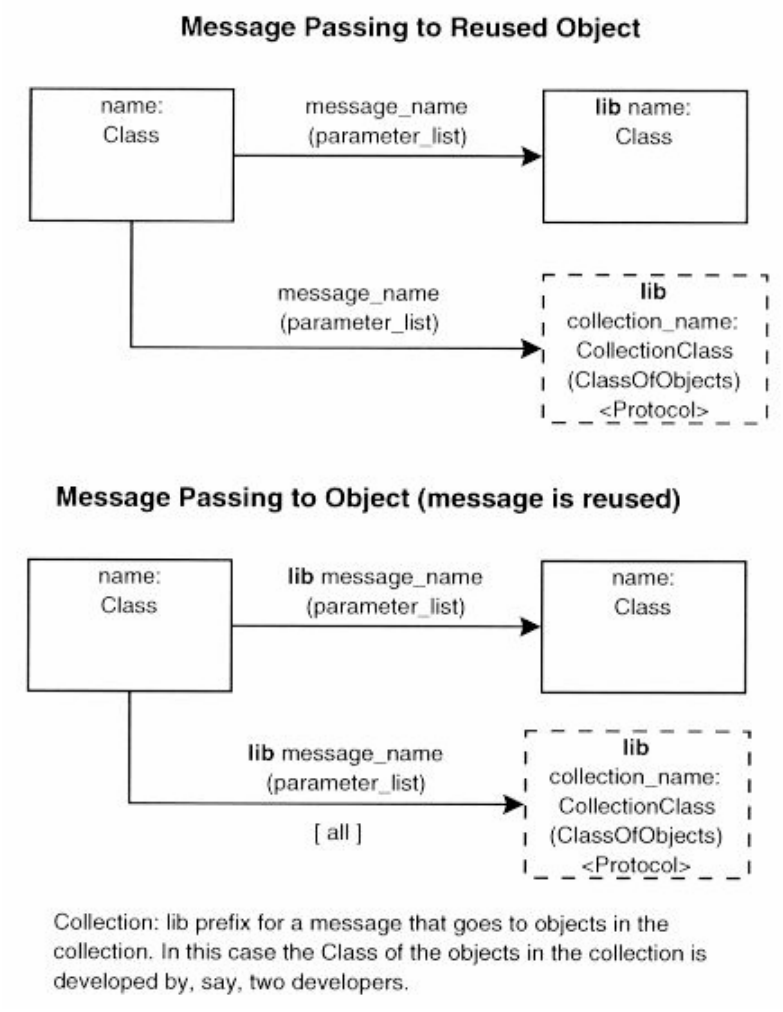


Figure 4 The OIG notation using **lib**

Figure 5 show an example. We derived this example, from Figure 4-10 page 72 of the first Fusion book (Coleman et al., 1994).

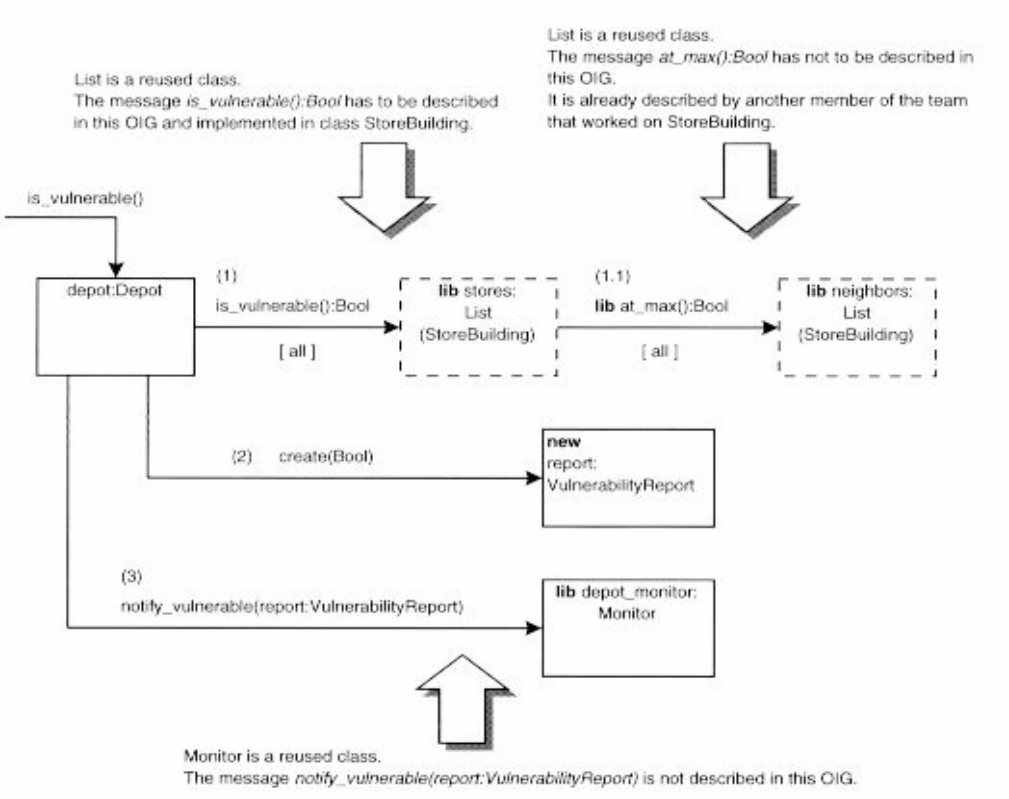
For *stores* and *neighbors* a List was used to store StoreBuilding objects. Because List is part of the Class Library, the developer gave it the **lib** prefix. The message *is_vulnerable():Bool* to *stores* is sent to all objects of class *StoreBuilding* in that list. The developer still has to add the method description (**method** StoreBuilding: is_vulnerable():Bool) to the OIG. The message *at_max(): Bool* was already described (and implemented) by another developer, hence the **lib** prefix for the message.

The class *Monitor* is reused. It therefore has a **lib** prefix so it will not be described in this OIG. A reference to where to find its description is, of course, very useful.

We discovered that, nine times out of ten, collection classes are reused classes. This means we do not build them ourselves. These collections are for example: List, Set, Bag, NSDictionary, Array, etc. all reused from the class library or the class foundation framework.

In the end everybody was using these notations. They know: “Resistance is futile, You have to do design in a way that makes life easier and less filled with defects.”

So far the adaptations work fine. They are clear to the developers and reviewers, and now we can direct our discussions at serious ambiguities and gaps in the models.


List is a reused class.
The message is_vulnerable():Bool has to be described in this OIG and implemented in class StoreBuilding.
List is a reused class.
The message at_max():Bool has not to be described in this OIG.
It is already described by another member of the team that worked on StoreBuilding.
is_vulnerable()
depot:Depot
(1)
is_vulnerable():Bool
[all]
lib stores:
List
(StoreBuilding)
(1.1)
lib at_max():Bool
[all]
lib neighbors:
List
(StoreBuilding)
(2) create(Bool)
new
report:
VulnerabilityReport
(3)
notify_vulnerable(report:VulnerabilityReport)
lib depot_monitor:
Monitor
Monitor is a reused class.
The message notify_vulnerable(report:VulnerabilityReport) is not described in this OIG.

DECISION SUPPORT METRICS

by Kris Oosting

"I need to make a decision now! Give me the right information." Everybody wants to know everything, but almost nobody wants to perform the necessary measurements to get the information in order to know "everything." Is it easy to get in the first place?

Over a period of more than 8 years we have learned that software metrics are very valuable. They tell good stories and bad ones. Metrics are also more close to our "hearts" than we are willing to admit.

This article describes the different classifications of metrics that we encountered while measuring Fusion projects over the past 3 years and the "tool" that supports it—the Project Measurement Framework.

I want to know it all!

That's what you often hear when people want to start or have just started collecting software metrics. Just wait a few seconds and others will start to complain about the accuracy, privacy, amount, and usefulness of metrics. Of course! If metrics are enabling you to tell what is going on, it can also reveal what is not so optimal.

Well, that's what metrics should tell us, providing us with an opportunity to improve whatever we need to improve. The market is very competitive. We don't need complex metrics to tell us that.

We might note that when one says that the market is competitive, nobody asks what the metrics were and if the information was normalized...

If we want to know it all, what can help us? To clarify metrics collection and use we classified the (usage of) metrics in this article as follows:

- Observation Metrics
- Decision Support Metrics
- Statistical Metrics
- Metrics of Life

We did this in order to deal with questions like "are these values weighted and normalized?", "that's a very simple metric, probably it tells you nothing!", or "all very interesting Mr. Metrician, but... we don't believe you can measure this."

Actually the most valuable metrics are very simple.

There is one very simple "instrument" that we almost don't use any more, or don't trust. This is called intuition.

Observation Metrics

Observation Metrics are metrics that come from observing the process, a project, or a product. We just want to see what is going on without making any judgments or jumping to any conclusions. These metrics are, in general, not experienced as "frightening." Examples are time, number and type of tests, lines of code, number of abstract use cases, number of model classes versus support classes, etc.

Observation metrics can, of course, later be used to make decisions.

It can be very useful to classify metrics collections as observations when doing it for the first time. Most people will try to convince you that you have to collect metrics for years before they can be used for whatever purpose. However this is not true. I don't have to collect my car's average mileage per gallon of fuel for years to tell whether it is expensive to run or not.

Another example of an observation metrics is what we call the *Red Cheeks Metric*. We observe this one often when we ask during process assessments for example "tell me, how is the development process you're using, working?". We observed that questions in the area of development process and quality have a high *Red Cheek* count. Interesting!

While observing projects you will often discover trends and handy forms of presenting the results. You will also have the possibility to set up the *measurement framework* for future projects without being too threatening at the time for project members.

We found that while discussing the observations with the project team, they often gave good suggestions for what the outcome of the various observations could be used for. For example a scatter plot of Cyclomatic Complexity versus Lines of Code (not-commented) gave some clues about big objects being complex. But when a developer suggested that it was perhaps better to have in the scatter plot the Cyclomatic Complexity versus Depth of Nesting (nested if-the-else) it not only showed high complexities for high depth of nesting, but it also initiated a discussion about the use of polymorphism. Another interesting point is that we also observed interesting defect analysis results. For all defects that have their origin in code, over 40% had defect type *Logic* and occurred because something was *Wrong*.

Observations are nice and useful. Now we want to make some decisions using metrics.

Decision Support Metrics

Decision Support Metrics are used to make decisions. These metrics have to assist us during decisions we face on a real project. Note that we do not have the time here to collect metrics for a few years—then it will be too late. We

have to make a decision now and need everything we can get to support the decision making process. After the decision, the metrics can be thrown away. They will have served their purpose.

What kind of decisions, you probably ask. Examples are: “Are these designs too complex?”, “Will these designs create an implementation problem?”, “What will happen in 2 weeks when this trend in hours-per-phase will continue?”, “How many resources do we need to allocate for pre-release defect removal?”, “what design pattern is the best to use here?”.

During projects we often face the question if a design has to be redesigned or not. That’s a decision we have to make. Because it’s our project with our own peculiarities we cannot rely on industry averages of 1000 other projects done in the past 10 years. We have to decide now. So therefore we have to use metrics like service degree (coupling), percentage of reuse, number of system operations and their category (estimation), etc. to make the decision. Once the decision has been made, the values of those metrics are more or less out of date, because the design will change.

Observation Metrics are often used in decision making situations. Some metrics were collected for quite some time, but never seen as being important for making decisions. In practice we did benefit a lot from metrics concerning time. For a more extended explanation, please see the articles titled “Do we have the time?” in the Fusion Newsletters of April and July, 1996.

Sometimes we need information concerning a lot of projects. Then Statistical Metrics can be handy.

Statistical Metrics

Statistical Metrics are important, but they are not always as powerful than the company’s own Decision Support Metrics. Somebody told me that “Statistical Metrics are published to impress the only three people in the world that understand the publication”. Which indeed is often the case. We made that mistake in the beginning as well—trying to impress people. The reaction we heard the most when reporting that “measurements over 1000 projects conclude that the average complexity is 23.45678” was that their project was not one of them so the values do not apply for their projects. Back to square one.

These metrics are useful, but do not over-estimate their power.

Metrics of Life

We humans tend to collect a lot of metrics every day. Without knowing it we make all kind of decisions and assumptions based on these metrics. A few examples are:

Traffic Lights. For 4 months I measured the percentage of red and green traffic lights. The collection process was writing tick-marks on a list.

What did I want to prove? Well, during discussions with colleagues we thought that traffic lights are always on red when we are on the road. It turned out not to be as drastic as we thought it was. 57% of all lights actually forced me to stop. (If a traffic light is red and then turns green and red again while you, poor soul, are still waiting, it is counted only once.) A population of 1273 traffic lights freely participated in the test. There were so-called Green and Red days.

Then another said, “but it’s not the number of lights alone. It’s also the amount of time you lose.” Interesting. Let’s measure that as well then.

Time Spoilage. This metric indicates how much time we loose when waiting for a red traffic light and or traffic queues waiting for red traffic lights. My sense was that I always had to wait an enormous amount of time. One of my customer sites is a 1 hour drive and 15 traffic lights away. My gut feeling was that I lost at least 10 to 15 minutes of that trip waiting at lights. I asked this of others and they gave estimates between 10 and 20 minutes as well. Ok, let’s measure it with a stopwatch. I was surprised. It was between 3 and 6.5 minutes!

Here the collection of metrics demonstrated that my perception of time was very wrong.

What has this got to do with projects, you will ask. Simple, just ask yourself “what is my perception of time when making an estimate or when I schedule resources?”. You will recognize the traffic light example.

Yeah... but, the insecurity metric. Another example of a metric of life. During discussions about measuring the software development process I noticed that people often said “Yeah..., but...”. It turned out that these people were very insecure. Example: “Yeah this is all very interesting these coupling metrics, but I do not see how they can be applied on this project.”

From experience we can derive the following rule of thumb’: If there is a high “Yeah..., but” count during the meeting, then the likelihood of success of the metrics exercise reduces rapidly.

For example, suddenly time registrations are forgotten, documents are not presented to auditors, developers are too busy doing “something.” All kind of tricks are played in order not to deliver the measurements.

Note that the success of metrics collection and

reporting also depends on the company culture—whether you like it or not. On the other hand project managers and developers complain that they cannot control the project, but they are also not willing to provide the right measurements.

DeMarco said, and you should know this one, "You can't Control what you can't Measure".

"Yeah he's right, but we do object-oriented software development...".

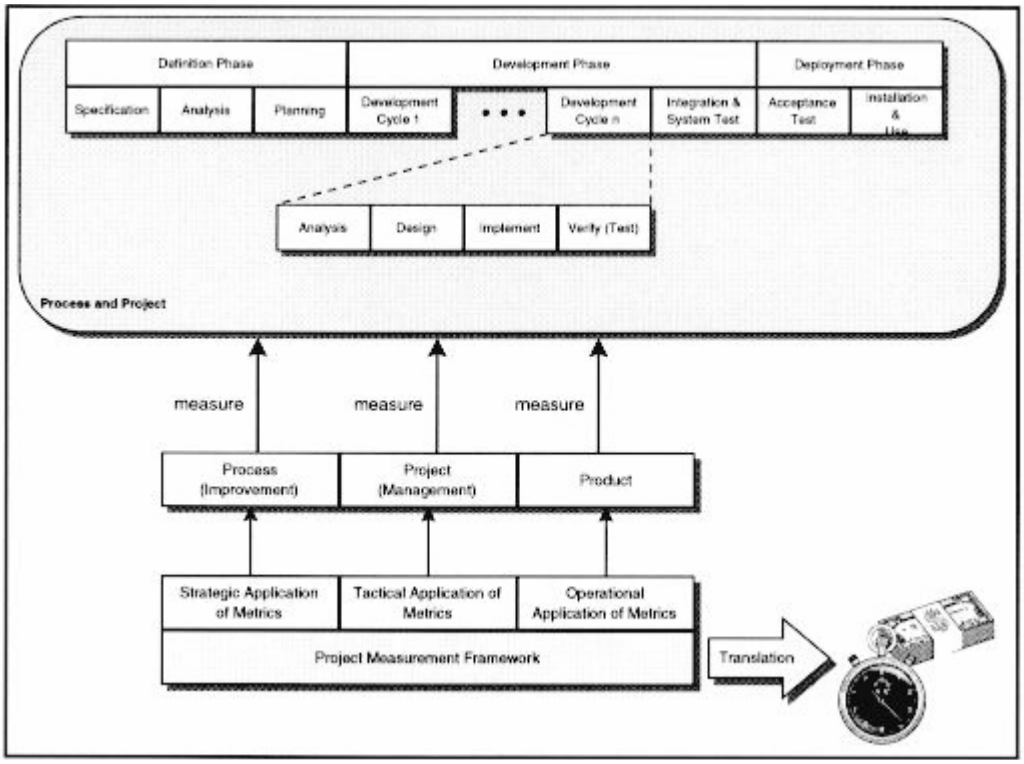


Figure 8 Project Measurement Framework (upper level)

Project Measurement Framework

The fact that the Fusion method has a good process and is easy to measure doesn't mean that measurements are taken. Often wrong parts are pointed out to be measured. For example, a product or a single project that has already gone very wrong. As Chinese doctors say to their patients: "that you're standing here with problems means that you're actually too late. You should have done Qi Gong (Chi Kung) exercises when you felt fine to prevent these problems".

Stay with this doctor for a second. I asked him: "If I come to you with a headache, do you give me an aspirin?". The doctor replied: "No! what you tell me is just a symptom of a problem. I can "cure" the symptom, but then in one week you'll be back. Let's see. Ah, this vertebra in your neck is a little off its place." Crraackk. "There, problem solved. That was the cause of your trouble!".

Me: "How come?"

Doctor: "Look, it's like software development. I've heard that there are still a lot of problems, defects they call them, in the software when it is delivered. People only seem to see the symptoms of the problems and companies spent fortunes to do something about the symptoms."

Me: "But there is almost no time to locate the cause and how can I find it anyway."

Doctor: "You have to perform the right diagnosis. You need to know how the patient's internals are working. We use all this equipment to measure certain organ's functions. I also compare the measurements with my database over here. If the patient was here before, I also compare the measurements with the earlier measurements to see if there maybe is a trend. If I discover something strange, then I can act in time. A kind of early warning if you wish."

Me: "I see. The patient's internals are our models of the software. Test reports is only symptom registration. Wait a second! There is something called Defect Tracking. Then you really look for the cause of the problem and try to cure it. Wouldn't it be nice if we could automate this somehow?" Thanks doctor, I have to run now!"

Doctor: "You're welcome. (big question mark on his face)"

Here were some good keywords: Diagnosis, Measurement, Symptom, Cause, Defect, Tracking.

What are we doing with them. Well, we will build a Project Measurement Framework. Figure 1 provides an overview of the framework

The process has three major phases: Definition Phase, Development Phase, and Deployment Phase.

Here a process is a description of procedures, etc. Projects use a process. Of course more projects can use the same process and add some project-specific issues to it. The project itself manages (contains) the definition, development, and deployment phase, for example.

Using this model we want to monitor three applications of metrics. They are:

- the strategic use of metrics—process (improvement);
- the tactical application of metrics—project (management);
- the operational application of metrics—product.

The Project Measurement Framework has to translate the information to *time* and *money*, the two things that decision makers understand and use as decision support metrics.

In a next article I will explain the three applications of metrics and what the Project Measurement Framework collects from the Fusion Process, Project, and Product.

We can control, because we can measure!
