On principles for model-based systems engineering

Ingmar Ogren, Tofs AB, Fridhem 2, SE-76040 Veddoe Sweden
e-mail iog@toolforsystems.com,
URL: http://www.toolforsystems.com
phone +46 176 54580
fax +46 176 54441

Published in "Systems Engineering Journal" Q1 2000

1. Abstract

This paper addresses the problem of consolidating technical descriptions of how a system is built with operational descriptions of the missions the system shall complete (how the system is to be used).
It also discusses how a central model constituted from design objects with requirements, test cases, problems and documents as attributes, to these design objects, can support modern principles for "incremental acquisition" and "incremental development".
Modeling principles, based on entity-relationship diagrams and the UML (Unified Modeling Language) component diagram, combined with pseudo code behavioral descriptions, are described as means to build the "central model".
After a "central model" for systems engineering is established ,it is shown how the model can be extended into a "Common Project Model", being common in two ways:

- *Common for "real implementations" and simulators required for the system.*
- *Common for all concerned stakeholders such as acquirers and contractors.*

Application of the "Common Project Modeling" principle, with computer-stored models, holds promises for increased system quality and for more efficient systems engineering.

2. Background with today's solutions

Leonardo daVinci once made some fantastic designs. From these you can understand that technical drawing, in that time, was little different from art. Later technical drawing diverged from art with the introduction of dimensional measurement, different views, and so on. This tradition of technical drawing has developed into qualified CAD drawing systems. Technical drawing is now an excellent means of describing the physical properties of any item, but it gives little understanding of how the item drawn should be used or which missions it can contribute to complete.

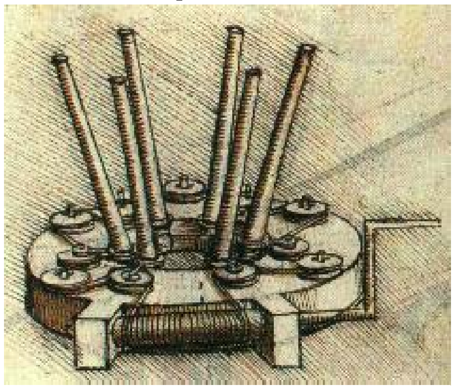


Figure 1 Leonardo DaVinci drawing

When electricity arrived, new drawing techniques were needed, resulting in the electric schema showing electric components, voltages and currents. Since electric equipment was often complex, the block schema was also introduced to show a higher "structural" level of the electric system. These schemas still concentrate on the system's components, giving little understanding of the system's missions.

With the advent of computer software, it was believed that many problems would be solved through the simplicity of changing the software. Rather soon this simplicity of change proved to be more of a problem than a solution and the need to describe software exactly was understood. One solution was "Structured Analysis" diagrams, derived from the earlier "block schemas". These give a good understanding of the software's structure, but still little information about the software's missions. During the last ten years object-oriented software descriptions techniques have become wide spread, mainly used as a means to support economic software reuse.

Modeling is a well-proven technique for technical research and development. Ships, buildings, airplanes etc. have been modeled for purposes, such as hydro- and aerodynamical research, usability investigation, visualization for end-users, etc. Modeling of software-intensive systems, using the Unified Modeling Language (UML) [1], introduced by Rational Inc, is now also possible. Models represent an excellent way to visualize one or more aspects of a system, but most models of complex systems still have problems in clarifying the system's missions. Below will be explained how a system can first be modeled in its context, after which the system's missions can be identified and included in the model. First however, a discussion of the development process structure.

3. The three basic processes in Systems Engineering

For software and systems engineering the combination of "waterfall" and "big bang" used to be popular. "Waterfall" then means that system development is visualized and planned as a number of time-separated phases, the main phases being analysis, design and verification. "Big bang" means that development is planned and executed as a single effort going through the phases from requirements' investigation to integration. The combination means that you plan system development as a single large concerted effort, composed from a number of phases, to be gone through, one at a time, separated by reviews. The principle is attractive for several reasons:

- It is simple to explain.
- It is orderly, logical and can be visualized in a single viewgraph.
- It is well suited to traditional acquisition with fixed price.
- It often offers an attractive time schedule, when presented in proposals.

However there is a small problem with the combination of "Waterfall" and "Big bang" for development and evolution of non-trivial systems, since the resulting methodology does not comply with reality. It simply does not work for reasons such as:

- It is not humanly possible to specify a complex system completely and correctly prior to development, since development will always build new knowledge.
- Problems will always surface during development and some of these will cause late changes of requirements.
- In reality, the activities in the "phases" are more concurrent than sequential, making it impossible to put them into a sequential schema.
- It is difficult and often impossible to know the real requirements, with their priorities, until end-users have had an opportunity to acquaint themselves with the final system or at least with a realistic representation (model) of the system.
- It is difficult (impossible) to know the "cost/contribution to mission" ratio for each system feature before getting rather far into development.

For these reasons several new development models have been defined for software engineering, such as "spiral"[2] and "ball-bearing"[3] models, with a better understanding of the need for concurrency. Several of these introduce new problems concerning acquisition as it becomes painfully obvious that you must understand that the principle of "fixed price contract" is of little use in complex system acquisition, when the requirements are not really known until you are well into development. One principle that gives promises to both manage the concurrency needed in systems/software engineering and to allow fixed price contracts is "Progressive acquisition" with incremental development[4].

A version of incremental development is visualized in Figure 2. The technique is characterized by:

- "Requirements Management", "Development" and "Verification with Test" are established as three concurrent processes.
- Successive releases of the system are produced, giving the developers and end-users "something real" to work with as soon as the first version is released.
- "Requirements Management", includes analysis and this process is more labor intensive in the beginning of the project

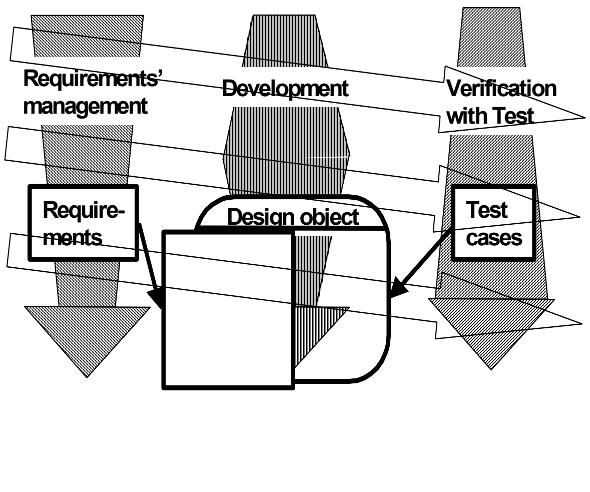


Figure 2 Parallel processes in incremental development

where most of the requirements' work still needs to be done.

- "Development" includes architectural and detailed design. This process is most labor intensive in the middle of the project.
- "Verification with Test" starts early in the project with verification of initial requirements, but is most labor intensive by the end of the project, with testing in connection with system integration and deployment.
- The three processes are kept together by a central design object structure with requirements and test cases being attributes to the objects.

The principles of incremental acquisition/development are now being introduced in some large system user organizations within the WEAG (The Western Europe Armament Group).

4. The central model

A problem with modern development models with concurrency and incrementality is that they tend to be confusing to Quality Assurance people. They don't find their traditional baselines and "critical reviews" and feel they are getting lost in a multitude of activities and versions.

This is a serious problem, which however can be made less serious through introduction of a central model to base the project on. There are several ways to model and it is essential to decide on modeling technique for a project.

4.1 How do you know what you model?

When you review a software or systems engineering diagram, you often come across simple entities such as for example "aircraft position". You can ask the diagram author what this means: "Is it really the aircraft position or is it the computer's understanding of the position?" The question may cause some confusion and most often the answer will be something like "It is this entry in the data dictionary, represented by that floating point data". If you then put the next question: "How do you know it is the real position?" you may get a clear, crisp and understandable answer. You may also get a confusing discussion of data, communication paths and delays throughout the system, which leaves you with little understanding of how well the data represents its counterpart in the real world.

In these cases it helps to draw a UOD (Universe Of Discourse) diagram. This is a simple way to increase knowledge of how entities in a system represent and connect to entities in the "real world".

To draw a UOD diagram, start with an entity in the real world, such as an aircraft:

Figure 3 A single entity

Next, you can introduce a radar to detect the aircraft together with a couple of relations between the radar and the aircraft. The relations are drawn "both ways" to show that this is not a Data Flow Diagram, but an Entity-Relationship diagram, which simply defines entities and relations. To read and understand the diagram, you simply read the text in one box together with the text along an arrow and the text in box the arrow points at. When you review a UOD diagram, you read these simple texts and check that they are both readable and say something meaningful about the system.

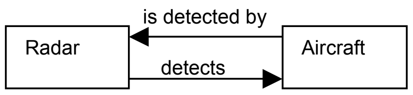


Figure 4 Two entities with relations

If you then want to build an air traffic control system you need to represent the aircraft in the system:

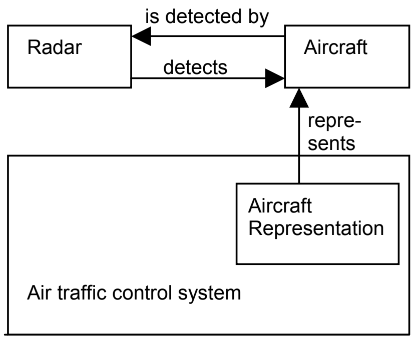


Figure 5 Representation entity added

What you have done so far is simply to add to the original diagram to show that radar is used to detect aircraft and that aircraft must be represented in air traffic control systems. This may seem completely trivial, but establishment of basic facts like these may well be of crucial importance in other and more complex circumstances.

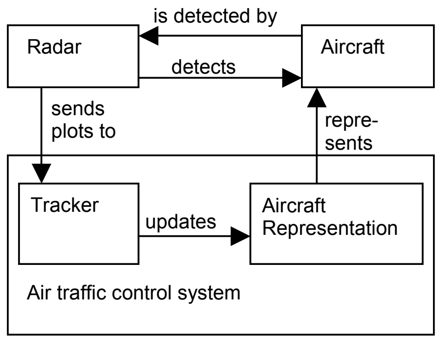


Figure 6 Diagram with "double coupling"

However, the diagram says nothing about how to build the "Aircraft Representation". For the example is presupposed that "Aircraft Representation" is built by a "Tracking" entity, which gets information from the radar. The entity "Tracker" is introduced, with its relations in Figure 6.

You now have a simple UOD diagram with "double coupling". The diagram shows an entity in the environment or "real world" (Aircraft) and its representation in a system (Aircraft representation). The double coupling means that the diagram shows how the environmental entity is represented in the system and how the environmental entity influences its representation.

The diagram also expresses a number of simple facts about the system in its environment if you read the text in each two connected boxes together with the arrow text. If these sentences don't make sense or are not grammatically correct, the diagram probably needs some further work.

What you have done now is basically modeling on two levels. The diagram is a model of a system in its environment and the diagram shows one aspect of how the system's environment is modeled within the system.

Note that "environment" is not necessarily the "real physical" environment. For example an embedded software system may well have other software systems as its environment!

UOD diagrams can be drawn simply with paper and pencil or on a blackboard and this is often an excellent idea, particularly early in system analysis, when you want to build an understanding of an existing or future system. Drawing these diagrams together with an experienced end-user on a blackboard is a very good way to understand and document basic facts.

However remember that what you are drawing is entities and their relations, not a data flow diagram and don't make the diagram too complex. Multiple small simple diagrams are better than one big complex diagram, since it will be difficult to see the errors in a complex structure.

Tooling is an issue for the UOD-graphs. The blackboard is a wonderful tool, but it has its limitations as a means for persistent information storage. Computer storage is better and many simple drawing programs, such as PowerPoint or Visio can be used to draw and store UOD diagrams. You can also use other programs with drawing capacity, such CAD or CASE programs.

However, before you select a program to document and store your models, check that it does not have any awkward syntactical limitations and that it can do useful tricks such "rubberbanding" and "snapping".

4.2 Requirements on modeling and modeling alternatives

From the above discussion it is obvious that modeling is central to achieve quality in complex systems. There are many ways to model a system and you may wonder which one to choose. The answer is very simple: It depends. It depends on which aspect of your system you want to model and who shall read your model. Another answer is that you cannot really choose, since you need to master a palette of modeling techniques to cover the needs during a systems' engineering effort. You must consider what is required for modeling a complex system, to achieve an acceptable quality. Five key requirements are:

1. Determinism with formality

This means that everything expressed in the model must have a single, defined and obvious meaning.

2. Understandability

Since systems engineering should be done in close cooperation with end users, the models used must be readily understood, without extensive education or experience in software or mathematics

3. Inclusion of system missions
The model must elicit the system's missions and also be able to express how different parts of the system contribute to completion of these missions.

4. Modeling of structure and behavior
The modeling technique shall support splitting a system into subsystems, with clarification of interfaces between these systems, and the modeling technique shall also allow definition of behavior within the subsystems defined.

5. Possibility of verification support
It shall be possible to verify a completed model. This verification can be against defined requirements, but it can also concern verification of completeness, consistency, etc. For complex systems, verification will often require computer support, depending on the large amounts of information to be managed.

4.3 Some modeling alternatives

Below some useful techniques for modeling are discussed and three of them are also shown in Figure 7.

The Block diagram
The block diagram may be the oldest way to model systems. It is very simple to understand and it can include events as shown in Figure 7. The example shows that A contains B and that B transfers something to C. The block diagram is extensively used for hardware schemata, for organization diagrams and for software structuring (as Data Flow and Context diagrams)

The UML Class Diagram
The UML (Unified Modeling Language) [1] contains a "Class Diagram", which concentrates on a system's components with class inheritance, dependency, association, aggregation and cardinality. The rich syntax and the great power of expression are obvious advantages for the UML Class diagram.

The rich syntax may also be a disadvantage since it is easy to draw complex and confusing diagrams when you use the full syntax. A good idea, particularly when you work with end-users, is to limit each diagram to a subset of the Class diagram syntax.

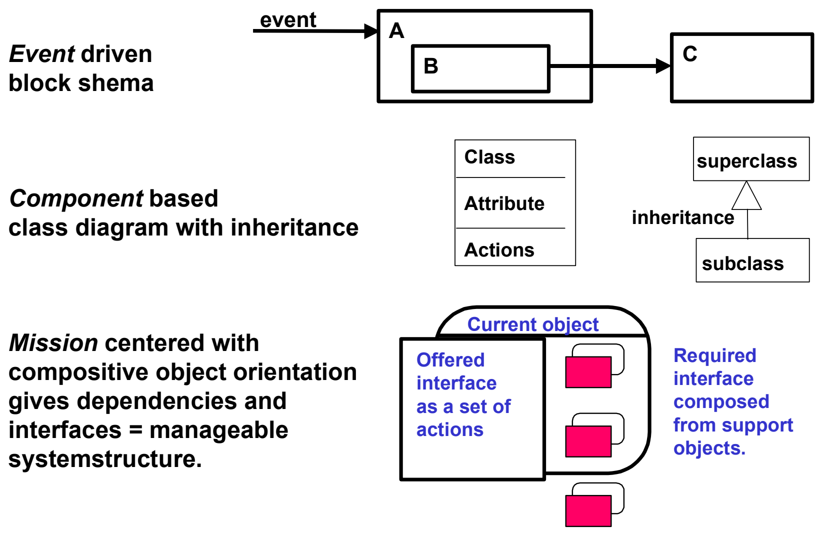

Event driven block shema
event
A
B
C
Component based class diagram with inheritance
Class
Attribute
Actions
superclass
inheritance
subclass
Mission centered with compositive object orientation gives dependencies and interfaces = manageable systemstructure.
Current object
Offered interface as a set of actions
Required interface composed from support objects.

Figure 7 Three principles for modeling

The UML Component diagram
Another UML diagram is the Component diagram. It was published in Grady Booch's book on Software Engineering from 1983 [5] and it has been developed in the HOOD software development method [6].

The UML component diagram is useful, since it can be used to model compositive object structures. When you work with such structures, you concentrate on each object's interfaces and on dependencies between objects, rather than on "inheritance" between objects.

The diagram in Figure 8 shows that:

- The current object has an offered interface (constituted from a set of actions, which can be invoked from outside the object)
- The current object has a required interface, constituted from parts of the offered interfaces of the support objects
- Two support objects are contained in the same system as the current object, while one support object is outside of that system.

The Component diagram allows you to model not only hardware and software components as objects, but also operator roles and missions. This makes it possible to model complex systems as a set of diagrams on different levels, with clear dependencies among the objects and with a clear understanding of how the different objects contribute to completion of the system’s missions.

	Determinism	**Understan-dability**	**Mission inclusion**	**Structure/ behavior**	**Verification support**
Block diagram	No	Very good	No	Structure	Poor
UML Class diagram	Yes	Difficult	No	Structure	Poor
UML component diagram	Yes	Good	Can be	Structure	Can be good
State charts	Poor unless formalized	Good	Can be	Behavior	Good if formalized
Pseudo code	Yes	Requires explanation	Can be	Behavior	Good if formalized

Table 1 Modeling techniques versus requirements

State Charts
State charts show behavior for a system component as a set of states with transition conditions and transitions between those states. Consequently, they can be used for modeling of behavior.

Pseudo Code
Pseudo Code is a code-like behavior description, constituted from code-like formal control structures, variables (parameters, messages and local variables) of defined types and comments.

Although the modeling techniques discussed here are only a small subset of the available techniques, it is still useful to compare these techniques with the requirements listed in section 4.1. What you need to model a complete complex system is at least one structural modeling technique and one behavioral modeling technique. The requirements and the modeling techniques are listed in Table 1. One choice (highlighted in Table 1), used for the continued discussion, is to start out from the UML Component Diagram and combine it with Pseudo Code.

This yields a central model, which defines system structure as a set of objects, depending on each other and connected through defined interfaces. Within each object its behavior is modeled as pseudo code in a set of actions.

In order to be able to manage the complete development effort attributes are added to the objects and used to manage requirements, test cases, problems and documentation.

4.4 The air traffic control example

In the Air traffic control example used in section 4.1 the initial UOD diagrams result in an understanding that one of the missions in this system is to manage aircraft information. Consequently an object “Manage_Aircraft_Info” is identified with sub-missions to present the aircraft information, to measure course and speed and to calculate collision risks.

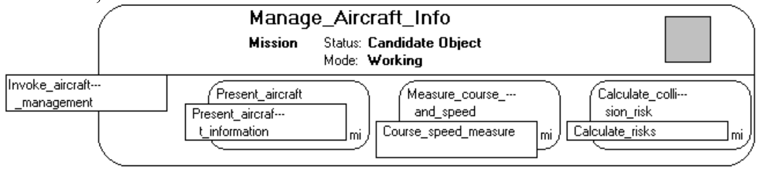


Figure 8 Object graph example from the Air traffic control example

Figure 8 shows the example in a modified UML component diagram, drawn with the Tofs toolkit [3].
The behavior description of the single action “Invoke_aircraft_management” in Pseudo Code will then contain one single concur statement to invoke actions in the three sub-mission objects in parallel

```
object Manage_Aircraft_Info is


action Invoke_aircraft_management is
visibility: Offered
purpose: {Invoke concurrent actions in mission
objects to complete the mission of managing
the aircraft information.}

begin
   concur
      # Calculate_collision_risk.
      Calculate_risks

      # Measure_course_and_speed.
      Course_speed_measure

      # Present_aircraft.
      Present_aircraft_information
   end concur
end

end Manage_Aircraft_Info
```

5. Use the model in the Three Basic Processes

The three basic processes in systems engineering are separate, but still connected through the central model. Below the content of the three processes and how they can be supported by the system model is discussed.

5.1 The Requirements management process

What to do

The requirements management process aims at creation and maintenance of an understanding of the requirements for the current project through the complete project. The requirements must be as complete as possible and they should comply with any constraints concerning, for example, scheduling and cost.

The need to maintain and optimize requirements makes it impossible to have the requirements process "done with" in the beginning of the project.

How to do it

The first thing to do to get the requirements correct is to find, understand and document the mission(s) of the system to be updated or created. After the missions are defined two things can be done in parallel: Define and assign requirements to the missions and create a draft system structure.

After this is done you have a basis to start the design and verification processes while the requirements process continues with addition of new requirements resulting from build-up of knowledge, adjustment of requirements as a result of problem management and distribution of requirements to design objects.

How the model supports requirements management

As soon as you have the missions and a draft top-level design, you can identify a first set of objects in the model. The requirements, problems, etc. can then be assigned as attributes to these objects. The result is that the model supports an orderly management of requirements and other pieces of information, which pertain to requirements management.

5.2 The Development process

What to do

The development process contains architectural and detailed design, expressed as a structure of connected objects. Each object will then contain detailed information to be used as a basis for implementation of that object.

How to do it

After you have the missions and the top-level requirements, you can apply top-down and bottom-up principles for design:

- Top-down through definition of new support objects to the mission objects, with continuation of the process downwards with distribution of requirements to the objects.
- Bottom-up through identification of reusable support objects with insertion of these in the designed structure and distribution of requirements to the objects found.

How the model supports design

During design the model is updated to include the new objects, defined or found, with their dependencies and interfaces. Consequently the model grows during design, and provided that your model is formal and computer-stored, it will support consistency checks of the design.

5.3 The Verification process with test

What to do

Verification includes verification of correctness for requirements, design and also for the completed and integrated system in its application environment. The fact that verification must be applied already on the first set of requirements makes it necessary to start the verification process in parallel with the other processes.

How to do it

Verification is done through reviews and tests on various levels and concerning various parts of the system under development or update. On the top (mission) level the system must be validated against scenarios, covering the missions defined.

How the model supports verification
When you have a computer-stored object model of a design, this model can support review work through presenting the design line-by-line for inspection and through automatic analysis of the designed structure.
Such a model further supports testing through allowing you to define test cases and test results as attributes to the objects in the design.

6. Managing the issue of criticality

Systems may be critical in different ways, for example safety-critical, mission-critical or environment-critical. For critical systems, you need an extremely low probability of failure. To achieve this it is helpful to have:

- A formal system description to allow for automatic checks on consistency and completeness.
- A system description which is understandable to the system's end-users, since these are the real experts on the system's applications.
- Fault-tolerance, since you must always allow for component failures and human mistakes in the completed system.

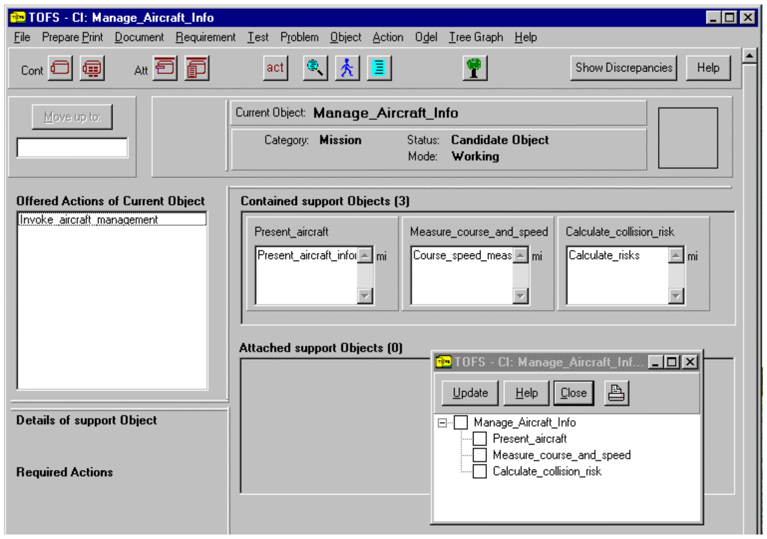
A model-based approach, as described above, helps to manage critical systems, since the model will support the necessary analysis activities in several ways:

- The formalized structural and behavioral system description gives the necessary basis for criticality analysis.
- Providing behavior is expressed in simple state charts or "English-like" pseudo code, the behavior should at least be explainable to end-users.
- The model gives an excellent basis for fault-tolerance analysis, since the model includes the dependency structure necessary for application of traditional analysis techniques, such as Fault Tree Analysis (FTA) and Failure Mode and Effects Analysis (FMEA).

7. Tools for modeling

Complex system are distinguished by the fact that it is not humanly possible to overview the system and at the same time keep an understanding of all the details of the system. When working with such systems, it is obvious that, provided that you have the ambition to find and retrieve the information, the amount of information exceeds what is possible to keep in mind or to manage as "paperwork". This is where a computer-stored model can assist. Stored in a suitable tool, such a model will assist in inputting and retrieving the large amount of information needed for work with complex systems.

One such tool, which supports the combination of component diagrams and formal pseudo code behavioral descriptions, is Tofs (Tool For Systems) [3]. For the discussion below Tofs is used for the example. A Tofs screen, with part of the Air Traffic Control example, is shown in Figure 9.


TOFS - CI: Manage_Aircraft_Info
File
Prepare Print
Document
Requirement
Test
Problem
Object
Action
Odel
Tree Graph
Help
Cont
Att
act
Show Discrepancies
Help
Move up to:
Offered Actions of Current Object
Contained support Objects (3)
Attached support Objects (0)
Details of support Object
Required Actions
TOFS - CI: Manage_Aircraft_Inf...
Update
Help
Close

Current Object: Manage_Aircraft_Info
Category: Mission
Status: Candidate Object
Mode: Working

Invoke_aircraft_management

Present_aircraft — Present_aircraft_info
Measure_course_and_speed — Course_speed_meas
Calculate_collision_risk — Calculate_risks

Manage_Aircraft_Info
- Present_aircraft
- Measure_course_and_speed
- Calculate_collision_risk

Figure 9 Tofs screen with component diagram for the object "Manage Aircraft info"

8. The Common Project Model (CPM)

8.1 The quality problem

Quality for complex systems concerns compliance between system performance and expectations. These expectations take different form for different stakeholders. For example:

- A simulator user expects the simulator to include a correct representation of the simulated system and its environment
- A system end-user expects the system to comply with his or her original specifications and with its documentation
- A system maintainer expects a system to be delivered with a complete and understandable documentation, which complies with the system.

All these expectations concern the fact that a complex system encompasses not only the system itself, but also a set of models such as simulators for different purposes, models included in the maintenance documentation and mental models maintained in the minds of developers, end-users and maintainers.

It is obvious that whenever one of these models deviates from the real system, a risk is introduced that one of the stakeholders has expectations, which deviate from the system's reality. This results in a quality problem.

8.2 Commonality Acquirer/contractor

Each stakeholder and participant in a project has the right to expect an understandable description of the part of the system he or she is concerned with. This description shall include not only design information, but also valid requirements, test cases, etc. For a partial system description, to be of acceptable quality, for a project participant, it must be possible to show that it is part of a consistent description of the complete system. As a complex system, by its very nature, is not completely understandable for one person at one time it is not an easy task to achieve the desired description quality.

Descriptions of complex systems present a number of problems:

- Modern complex systems are composed from multiple sub-systems, which must cooperate in order to complete missions.
- Subsystems are delivered from different vendors and are typically utilized by end users in separate organizations.
- Systems operate in a complex environment, in which external systems may influence mission results.
- Systems will normally exist in multiple releases and may be supported by several simulators, each providing a model of the system.
- Legacy and COTS (GOTS) parts must be integrated into the system with full understanding of how they are interfaced and of how they contribute to completion of the system's mission
- The different vendors and end users, concerned with a complex system will each have their own standard and tradition for system descriptions.

The problem aspects can be summarized as follows: It is crucial to project success and quality that a common understanding covering all system releases and simulators be established between all of the involved parties.

8.3 Why a Common Project Model

As discussed above, a quality problem will surface whenever multiple expectations and models are present in connection with a complex system. It is obvious that it would be possible to diminish these problems if everyone concerned with a complex system could work from a common model to get a common understanding of the system and consequently also common expectations.

8.4 What is a Common Project Model?

A Common Project Model (CPM) is a description of a system's structure and behavior, expressed in a way that manages the problems listed in section 8.2. For any non-trivial and complex system it will be necessary to have the model computer-stored to manage and analyze the large amount of information required. Since a complex system will include a lot of documentation on parts of the system, it will be necessary for the model to include references to various pieces of documentation.

8.5 How to build a Common Project Model

To build a CPM, you should start when the project is in its concept stage. The model can then grow together with the project through analysis, design, implementation and commission. It is also possible to build a CPM for an existing project and consequently create the necessary common understanding from existing documentation and from the stakeholders' knowledge and expectations.

To build the model, you can use any qualified systems engineering tool, which includes acceptable modeling principles as discussed above. As an example is shown, in Figure 9 a Tofs screen picture of the initial structure for the "Manage Aircraft info" of the Air Traffic Control example.

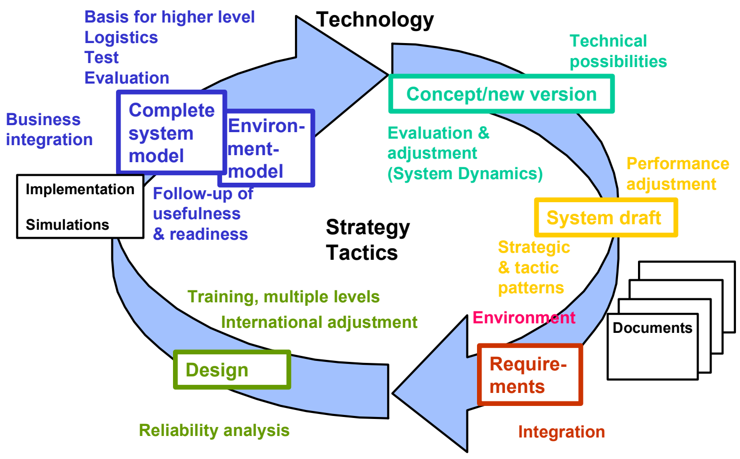


Figure 10 The Common Project Model as a "Project backbone"

Note that a tool to manage Common Project Models must include sub-tools to manage requirements, test cases and documentation as shown in the menu bar in Figure 9.

8.6 How to use a Common Project Model

When a CPM is built through a project it can be seen as "project backbone" as visualized in Figure 10.

The CPM can be used to support a variety of important activities to raise the overall system quality, for example:

- During development, the model helps to establish a common understanding of the system's environment and the environmental requirements on the system.
- During marketing of new system releases the CPM helps to make it possible to model new environments and to study how the system can be tailored to meet the requirements from these.
- In subcontracting, the CPM helps to ensure a common understanding among the contractors involved.
- In system maintenance the CPM helps to provide the necessary understanding of how the different parts of the system contribute to completion of the system's missions.

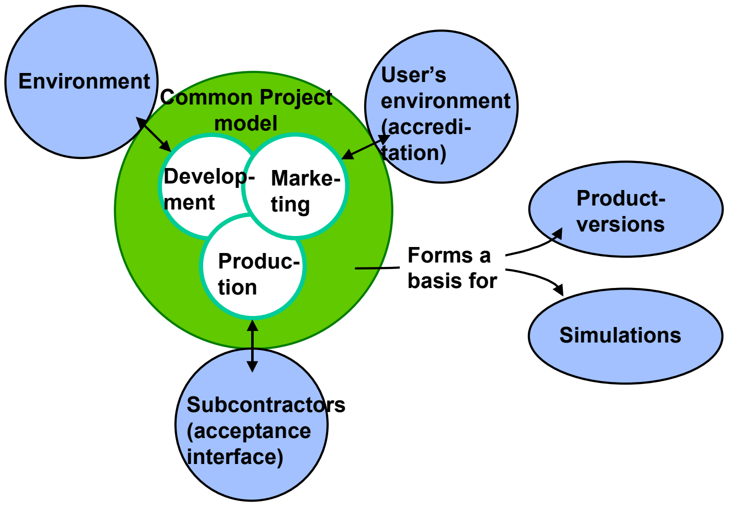


Figure 11 Use of the Common project Model

9. Experiences

The modeling principles, described in this paper, have been applied in multiple projects in

Sweden, primarily in defense and industrial applications. Some experiences are:

Application of parallel processes in incremental development.

Particularly early introduction of reviews, for verification of requirements has proven to be an efficient way towards early detection of requirements-related problems.

Inclusion of Mission objects in system structures.

Early identification of system missions and inclusion of these missions in the system structure has proven valuable to build a common understanding of the objectives for a system development among the stakeholders concerned.

Use of Universe Of Discourse diagrams

The Universe Of Discourse diagrams have proven to be valuable to clarify a system's interfaces with understanding of how the system represents its environment.

Use of UML Component diagrams (Object graphs) to structure models

These diagrams are not as easily understood as traditional "block schemas", why many developers are hesitant to use them. On the other hand the component diagrams have proven their usefulness to build system models to connect a system's misisons with all its components in a single consistent structure.

Use of Pseudo code to formalize behavioral descriptions

End users are hesitant to read pseudo code although it has been proven possible and useful to explain pseudo code to end users. Consequently pseudo code represents a useful compromise between informal natural language an d formal mathematical notations.

Use of Common Project Models

The idea of a Common Project Model (CPM) originated during development of Saab's PMSIM (Presentation and Control simulator) [7] and was to some extent applied in that project. No major CPM has yet been built but the principles have attracted interest form multiple major aerospace industries and the Defense Material Board in Sweden why a project for investigation of CPMs is under way.

10. Conclusions

1. The problem of modeling a system to show, not only its technical structure, but also how the system's components contribute to completion of the system's missions can be solved through extended application of UML component diagrams.
2. The present trend towards incremental acquisition and development, requires parallel processes for requirements management, development and verification. It is possible to achieve the necessary common basis for the three processes with an object based system model.
3. Modeling of system behavior with pseudo code gives a practically useful compromise between informal natural language and mathematical formalism.
4. Introduction of "Common Project Models" is a promising technique to increase system level quality and efficiency in evolution of complex systems.

11. The author

Ingmar Ogren was graduated with an M SC in Electronics from the Royal University of Technology in Stockholm in 1966.
He then worked with the Swedish Defense Material Administration and various consulting companies until 1989 with systems engineering tasks in areas such as Communications, Aircraft and Command & Control.

He now chairs the board and is owning partner in two companies: Tofs which produces and markets the Tofs (Tool For Systems) software and Romet, which consults in the area of systems engineering methods with the method O4S (Objects For Systems) as its main product.
Further information about Ingmar Ogren can be found on the web page http://www.toolforsystems.com

12. References

[1] UML, The Unified Modeling Language, information available from http://www.rational.com

[2] Information about the spiral model, originated by Dr. Professor Barry Boehm can be found at and downloaded from http://sunset.usc.edu/WinWin/winwin.html

[3] Information about the ball-bearing model for system evolution, as well as the Tofs toolkit software, can be downloaded from http://www.toolforsystems.com

[4] Guidance of the Use of Progressive Acquisition, © 1997 WEAG TA-13 & Contributing Companies
[5] Grady Booch: Software Engineering with Ada, Benjamin Cummings 1983
[6] HOOD, An Industrial Approach for Software Design, Jean-Pierre Rosen, Hood Technical Group 1997
[7] Saab PMSIM, Information available from http://www.saab.se