Does Object-Oriented System Engineering Eliminate the Need for Requirements?

Joseph Kasser D.Sc., C.Eng., CM
Systems Engineering and Evaluation Centre
University of South Australia
School of Electrical and Information Engineering F2-37
Mawson Lakes Campus
South Australia 5095
Telephone: +61 (08) 830 23941, Fax: +61 (08) 830 24723
Email: joseph.kasser@unisa.edu.au

Abstract. This paper examines system engineering (SE) and object-oriented (OO) methodologies and then shows both that SE is inherently OO and that OO languages such as the Unified Modeling Language (UML) may be used to document the user's needs in a manner that can be used by developers. The paper also suggests a next generation tool concept that can be used to hold both user and developer representation of the user's needs as an alternative to, and an improvement on, text mode "requirements", hence increasing the reliability of the shared meaning of the user's needs amongst all stakeholders.

INTRODUCTION

Kasser and Schermerhorn (1994) wrote that systems engineers needed to use a methodology that seamlessly interfaced to the software development methodology to avoid delays and errors in translation from the system requirements to the design phase. So as software engineering (SWE) has adopted an object-oriented (OO) perspective, system engineering (SE) has attempted to do the same. However, at the same time, SWE seems to be discovering for itself, the advantages of using the functional decomposition approach coupled with an OO perspective (Chang and Hua, 1994). This paper presents a brief analysis of both SE and SWE, shows that SE is inherently OO, and then suggests ways that three elements of the OOSWE paradigm may be used to enhance SE. Furthermore, when these elements are coupled with a 'yet to be built' next generation requirements tool text-mode based requirements may become an academic issue.

SYSTEMS AND SYSTEM ENGINEERING

SE has traditionally been viewed as following a process known as the Systems Development Life Cycle (SDLC) to bring a system into existence. Systems engineers are the people who implement the SE process. Blanchard and Fabrycky (1981) provide the following definitions.

System - an assemblage or combination of components or parts forming a complex or unitary whole.

Elements of a system - Systems are composed of

1. **Components** - the operating parts of a system consisting of input, process, and output. Each system component may assume a variety of values to describe a system state as set by control action and one or more restrictions.
2. **Attributes** – the properties or discernible manifestations of the components of a system. These attributes characterize the parameters of a system.
3. **Relationships** – the links between components and attributes.

Function – the purposeful action performed by the system.

Systems and subsystems – Every system is made up of components, which can be broken down into smaller components. If two hierarchical levels are involved in a given system, the lower is conveniently called a subsystem.

Blanchard and Fabrycky also state that SE *per se* is not considered as being an engineering discipline in the same context as the technical specialties it represents. The need for SE is present because many of the engineering specialists in one or more of the conventional engineering areas do not have the experience or knowledge to consider the system as a whole as well as knowledge of the non-functional technical specialties.

The goal of SE (Kasser 2000a) is to provide a system that:

This work was partially funded from the DSTO SEEC Centre of Expertise Contract.

- Meets the customer's requirements as stated when the project starts.
- Meets the customer's requirements as they exist when the project is delivered.
- Is flexible enough to allow cost effective modifications as the customer's requirements continue to evolve during the operations and maintenance phase of the system life cycle.

In the traditional SE paradigm, the systems engineers partition the system by function such that each component of the system provides some of the desired capability. Or in OO terminology, they encapsulate the components of the system by localization of functions to produce a system. Moreover, Hambleton (1999) points out that system engineers do not always map functions to physical components in a 1:1 mapping. In the example of a bicycle, the front wheel (physical) can be considered as an element in at least two virtual systems (steering and drive).In actual fact, a system may be represented by many kinds of objects, appropriate to the analysis in process.

In practice, SE tends to focus on the "functional capabilities required" to build a physical system (Eisner 1988). The "non-functional capabilities" needed or "non-functional requirements" tend to be shunted aside during development causing later problems when the system "as-delivered" does not perform its intended task in its operating environment. This is ironic, because the physical "as-delivered" system has both functional and non-functional properties. The current systems and software acquisition and development paradigm is characterized by mind-boggling complexity, a turbulent transition from function-based designs to OO designs, cost and schedule overruns (CHAOS 1995; OASIG 1996), and project failures (Capers Jones 1996).

THE OBJECT-ORIENTED PARADIGM

While SWE claims to have invented the OO paradigm, it was documented as an analysis methodology at least as early as the 12th Century. According to Maimonides (12th Century), an object is characterized by its

- definition;
- part of its definition, namely what it inherits from a parent;
- attributes;
- relationships with other objects;
- internal actions.

Various viewpoints of objects. Van Vliet (2000) states that there are different viewpoints of what the notion of an object entails, and provides the following definitions.

- **The modeling (European) viewpoint** – an object is a conceptual model of some part of a real or imaginary world. Each object has an identity that distinguishes it from all others. Objects have substance: properties that can be discovered by investigating the object.
- **The philosophical viewpoint** - objects are existential abstractions fall into two categories
  - **ephemeral** - exist for a period of time, and
  - **eternal** - live forever.
- **The SWE viewpoint** - objects are data abstractions, encapsulating data as well as operations on the data. This viewpoint stresses locality of information and representation independence.
- **The implementation viewpoint** - a contiguous structure in memory (software); may be composite or aggregate (possesses other objects)
- **The formal viewpoint** – a state machine with a finite set of values, and a finite set of state functions.
- **In the problem domain** - objects are abstractions reflecting the capabilities of a system to keep information about it, interact with it, or both.
- **The solution viewpoint -** objects are encapsulations of attribute values and their exclusive services.

Van Vliet also notes that objects with the same set of attributes are in the same class. Individual objects in a class are called instances of the class. Objects encapsulate state and behavior where behavior is described in terms of *services* provided by the object, and services are invoked upon receipt of a message from another object. The behavior of an object is described by systems engineers in terms of the functions performed by the object, while the software engineer describes the same behavior in the terminology of "services provided by the object." Apart from the semantic difference in the use of the words "functions" and "services", Van Vliet's objects are systems and subsystems in SE terminology.

The OO paradigm, based on Structured Programming concepts, embodies its own terminology in three basic principles:

- **Encapsulation** - the concept of placing data and processes, or methods or routines that operate on that data together and combining them to create a structure (object) that contains both. In SE terminology **encapsulation is design**, and an object would thus seem to be a subsystem.
- **Inheritance** - the concept that new objects are derived from existing objects. These new objects can add to or vary their behavior with respect to the behavior of the parent objet. This is the main feature that can lead to re-use of existing software. Inheritance can be both physical as well as

logical showing up in the hardware world as well. For example, a new communications component will inherit characteristics from existing components.

- **Polymorphism** – the concept that causes different types of objects derived from the same parent object to be able to behave differently when instructed to perform a same-named method in a different implementation.

Van Vliet classifies OO design as a middle-out design method. The set of objects identified during the first modeling stages constitutes the middle level of the system. In order to implement these domain-specific entities, lower level objects are used. This concept is no different to the systems engineer designing a system using subsystems. The only difference may be in the scope of the design activity. Hardware engineers designing a printed circuit board (in general) do not design their own customized integrated circuit components but use commercial off-the-shelf ones from a manufacturer's range. Today's software engineer, working with components may have to design and construct their own components to provide capability not available in commercial off the shelf (COTS) components in order to build the software system.

DIFFERENT PERSPECTIVES OF SYSTEMS

Consider the process-product production sequence in the up-front activities of OOSE and OOSWE as shown in Figure 1. OOSWE generates objects directly from the *use case*s, while current OOSE first develops requirements and then develops objects[1]. Electrical engineers will recognize that Figure 1 shows the OOSWE approach short-circuiting the requirements stage of OOSE. However, OO Systems and SWE do have different perspectives of systems as discussed below.

SWE considers information flows, and functional requirements. Non-functional requirements tend not to be considered in the software design. In its early days, SWE tended to ignore the physical domain, since the software operation was on a single hardware platform, however with the advent of distributed systems and client server techniques, software operates across several hardware platforms and the system needs to be optimized as a whole.

According to Van Vliet, the OO approach to systems analysis and design involves the following three major steps in an iterative manner.

1. Identify the objects.
2. Determine their attributes and services. The functions performed by the system are represented by the services performed by the objects.
3. Determine the relationships between the objects.

SE. In addition to the information flows, and functional requirements, SE also has to consider the physical properties of a system and the non-functional requirements. SE employs different models or perspectives for different purposes. For example, Hatley and Pirbai's (1994) methodology employs three models (Requirements, Architecture and Specification) and Menzes et al. (1999) discus viewpoint-based requirements engineering and its advantages. In SE, each model abstracts attributes so that only the aspects being studied are seen or in OO parlance uses the concept of "information hiding." For example, when considering the characteristics of a laptop computer one model may consider the physical aspects of cabinet, keyboard, disk drives, etc.; a second may consider the underlying hardware architecture of the computer; and yet a third may examine aspects of heat transfer inside the cabinet. SE may use different objects in each model. Lagakos et al. (2001) writes that *"The object-model formulation views a system as a group of interacting objects that work together to*

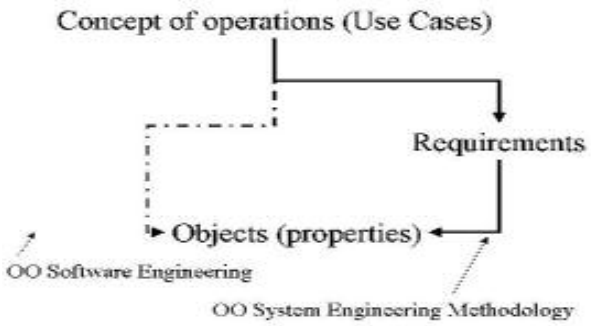

Figure 1 Process-Product Production Sequences

accomplish system objectives and satisfy system requirements. Use-case and domain models provide a visual representation for high-level system functionality and system design." If they had used the word "interacting sub-systems" instead of "interacting objects" they would have reiterated one of the maxims of SE in the language of traditional SE.

The traditional functional analysis and allocation can be described as following two major steps in an iterative manner.

[1]The Command, Control, Communications, Computers, Intelligence, Surveillance, and Reconnaissance (C4ISR) Architecture Framework Architecture Framework (C4ISRAF, 1997) approach also skips the requirements phase going directly from *use case*s to architectures.

1. Identify the functions performed by the system.
2. Partition the system such that each combination of one or more functions is performed by a physical subsystem.

Similarity in analysis and design. The same two factors are implicit in both SE and OOSWE, namely

1. The designer determines that the relationships between the objects/subsystems are such that the emergent properties of the system when constructed meet the needs of the customer.
2. The system is to be designed with minimal coupling between subsystems/objects and maximum cohesion within a subsystem/object, namely good engineering practice.

Thus SE is inherently an OO approach, however, the perspectives of SE and OO software are different. Sommerville (1995) states that the initial stage of function-oriented design relies on identifying functions which transform their inputs to create outputs while the system is treated as a black box. The OO design focuses on the entities within the system and considers the functions as part of the entities. Each design approach will generally produce different system decompositions. Somerville states that the most appropriate design strategy is often a heterogeneous one in which both the functional and OO approaches are used. This is not surprising as SE uses the identical approach in a 'design to inventory' situation to assemble a system out of components in the inventory.

Chang et al. (2001) propose a concept known as function-class decomposition (FCD). FCD, an iterative process, applies a top-down approach to decomposing a system while simultaneously identifying and grouping classes into functional modules, with each module representing a specific functionality which the system requires. Chang et al. seem to have discovered and applied SE!

ENHACING SYSTEMS ENGINEERING

In summary, while the OO approach is little different from traditional SE it explicitly emphasizes three elements that may readily be used to enhance SE, namely

- **Properties** which have values and attributes. Thus capability and requirements may be expressed as properties provided or properties desired.
- **Inheritance** which explicitly moves experience from designer into design. Whereas in SE, the non-functional aspects of a system were often a function of the experience and expertise of the designer, the concept of inheritance can be used to inherit (non-functional and other domain) properties and eliminate some of the "missing" requirements of today'' paradigm.
- **Encapsulation or design** which can be non-functional (i.e. data, and process) as well as the traditional functional approach to partitioning a system.

THE (PROCESS) INTERFACE BETWEEN SE AND SWE

The interface has traditionally been at either the System Requirements Review or the System Design Review. System engineers have focused on generating requirements to ensure that the as-built system is fit for its intended purpose. However, the current paradigm produces poorly written requirements (Hooks 1993) and various approaches have been proposed over the last nine years to alleviate the situation without much success. For example, Jacobs (1999) states that a 1997 analysis of the software development process performed at Ericsson identified *"missing understanding of customer needs*" as the main obstacle for decreasing fault density and lead-time. Related findings were aggregated under the heading "*no common understanding of 'what to do*'". The counter measures to overcome these problems focussed on testing the quality of the requirements rather than producing good requirements. There was no proposal on how to get clear requirements, nor was there a clear understanding of what a clear requirement was. **Thus in practice requirements do not seem to be a useful communications tool for translating between user needs and the system and software designers.**

IS THERE AN ALTERNATIVE TO "REQUIREMENTS"

If "requirements" are still poorly implemented after all these years, perhaps they should be eliminated or bypassed (automated). Kasser (2002) proposes a tool to improve the wording of requirements, but a greater degree of improvement should be achievable by replacing "requirements." Consider ways in which this might occur.

Requirements. Gabb et al. (2001) define a requirement *as "an expression of a perceived need that something be accomplished or realized.* Van Gaasbeek and Martin (2001) quote Dahlberg as stating that "**we don't perform SE to get requirements** and add "**we perform SE to get systems that meet specific needs and expectations**." The focus is on user needs, not requirements. What SE appears to have forgotten is that requirements are used to document user needs in a verifiable manner, **Requirements are a means, not an end**. There is nothing divine about requirements, they are just a convenient poorly-used

tool for translating user needs into a system that should be built.

Requirements are developed as an intermediate work product in the system development process, and are developed to provide formal communication between the stakeholders. Writing text-based unambiguous requirements for combinatorial and sequential scenarios in the form of imperative construct statements is difficult. Timing and state diagrams are often used within the context of the Requirements Document to provide the necessary information. Thus the concept of stating user needs (under certain circumstances) via diagrams is already in use in SE. Sutcliffe et al. (1999) proposed reducing human error in generating requirements by analyzing requirements using an approach of creating scenarios as threads of behavior through a *use case*, and adopting an OO approach.

THE UML 1.3 PERSPECTIVE

The Unified Modeling Language (UML) being a language has no inherent limitation on the number and types of objects, and thus there seems to be no reason why the UML should not be used to represent viewpoints or models at the system level as well as the software level. The UML (1999), perspective on complex systems can be summarized as

- Best approached through a small set of nearly independent views.
- No single view is sufficient
- Every model may be expressed at different levels of fidelity.
- The best models are connected to reality.

This perspective is little different to that of Checkland (1981) and Kline (1995). The UML is a modeling language for specifying, constructing, visualizing, and documenting, the artifacts of a software-intensive system. **The UML is not a process or a methodology**. This fact does not seem to be appreciated in the SE community, as for example, Gibbons (2001) writes *"The UML has provided a methodology that encompasses many of the up-front SE activities in an otherwise OO-based program."* This is because UML can be used to document SE products in those up-front SE activities within conventional SE methodologies. The UML diagrams for models cover

- *Use case*s;
- Classes;
- Behavior in terms of state, activity, and interaction;
- Charts - showing sequence and collaboration;
- Implementation, aspects of components and deployment.

The UML has a four layered architecture, which can result in systems being constructed from the center outward. The focus is on *use case*s, which drive the design. This is no different to the SE approach in which Kasser and Schermerhorn (1994) point out that the Systems and Operations Concept Document (OCD) is one of the two critical documents in the SDLC. Gabb (2001) summarizes the purpose of the OCD as describing the operation of a system in the terminology of its users. It may include identification and discussion of the following

- Why the system is needed and an overview of the system itself.
- The full system life cycle from deployment through disposal.
- Different aspects of system use including operations, maintenance, support and disposal.
- The different classes of user, including operators, maintainers, supporters, and their skills and limitations.
- Other important stakeholders in the system.
- The environments in which the system is used and supported.
- The boundaries of the system and its interfaces and relationships with other systems and its environments.
- When the system will be used, and under what circumstances.
- How and how well the needed capability is currently being met (typically by existing systems).
- How the system will be used, including operations, maintenance and support.

Gabb also provides the traditional SE perspective when he writes that "*an OCD is not a specification or a statement of requirement - it is an expression of how the proposed system will or might be used, and factors which affect that use. As such it is not obliged to follow the 'rules' of specification writing and can be relatively free in its language and format. Generally it will contain no 'shalls'*." However, there is nothing to preclude the use of *use cases* for describing all of Gabb's points in a verifiable manner in a manner understandable to both users and developers.

Lagakos, et al. (2001) state that a *use case* is simply a set of system scenarios tied together by a common user goal (i.e., aspect of system functionality), and describes a way in which a real-world actor would interact with the system. According to them, a *use case* specification contains:

1. A list of actors (actors are anything that interfaces with the system externally);
2. A boundary separating the system from its external environment;
3. A description of information flows between the actors and individual *use case*s;
4. A description of normal flow of events for the *use case*, and

5. A description of alternative and/or exceptional flows.

Gabbar et al. (2001) state that UML has been proven to be an efficient and comprehensive approach that can describe all three dimensions of the physical aspects of a production plant (static, behavior and function). However, Jorgesen (2001) writes, *"use cases do not replace requirements*" and Ogren (2001) points out that the UML is vague when it comes to requirements capture, and there is no precise definition of what a *use case* should look like. In SE terminology, **the UML states the requirements for a *use case*, but does not provide a design**. As stated above, the UML is a language not a methodology. As such it may be used for documenting requirements, but there is no inherent methodology for capturing them.

Limitations of Version 1.3 of the UML. Glinz (2000) points out nine deficiencies in the 800 page UML Version 1.3 and proposes modifications to overcome these deficiencies. Kobyn (1999) also discusses proposed modifications to Version 1.3, and IML Versions 1.4 and 2.0 are in process with products currently available on the OMG Web Site. Glinz also points out that the UML is built upon two fundamental concepts:

1. A class model is the core of a UML specification. This can easily be seen when analyzing the contents of the core package in the UML metamodel. The UML class concept still preserves the basic paradigm of its ancestor, the entity-relationship model: it is basically a flat structural description of the objects that the system has to deal with.
2. Specifications in UML consist of a collection of loosely coupled models (class model, *use case* models, collaborations, activity models, etc.). These are tied together by very few and semantically quite weak rules.

Glinz states that both these concepts are not at the heart of an OO modeling approach writing that "if we put them aside, we open a design space for OO modeling languages that are conceptually different from UML." He has developed such a language in his research group known as ADORA (Analysis and Description of Requirements and Architecture). The foundation of ADORA being a hierarchy of abstract objects, where each object integrates the aspects of structure, functionality, behavior and user interaction. Thus the upgraded UML, ADORA and other OO languages have the potential to describe both the user needs and the capability of a system.

REPLACING "REQUIREMENTS" BY PROPERTIES

So, if *use case*s can represent the user's needs in a manner verifiable by all stakeholders, then an improvement on the current text-mode based requirements paradigm will have been made. Kasser (2000) expressed "requirements" in terms of "properties and services needed" and capability in terms of "properties and services provided". The use of *use case*s driving an OO approach describing properties of components can provide the same representation of user needs as that of "requirements" if each property consists of an attribute and a value.

Consider "property" as the totality of the attribute and its value. Then requirements can be stated as the properties needed, and capability can be stated as the properties measured or exhibited by the object. The words functional and non-functional requirements no longer have to be used. When the system is broken down into subsystems each property (attribute and value) is allocated subsystem elements. Traceability of properties (functional and non-functional) is built into the approach.

The objective perspective shown in Figure 2 lists the properties of a component. Each property has an attribute, which has some value. In the example of a communications object, as far as performance attributes are concerned, the data input attribute has a value of 1000±10 units, the Data Figure Product flows output attribute a value of 1000±10 units, etc. The services (functions) performed by the object have to do with ingesting the data input, performing some action on the data and forwarding processed data. The non-functional attributes such as reliability (MTBF, and MTTR) and operating temperatures are

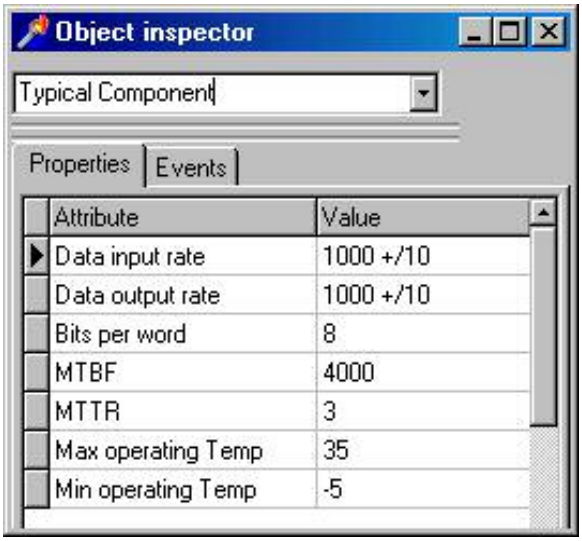

Figure 2 The Object View

also shown.

THE NEXT GENERATION OF REQUIREMENTS TOOLS

From a historical perspective, user needs, expressed as "requirements" were originally transmitted in requirements documents and specifications. As tool technology advanced requirements databases were employed and these documents became printouts or reports from the databases. Today's requirements tools tend to operate at this level. Systems and software engineers using *use case*s can employ tools using the appropriate representation provided by the tools at hand and a new tool.

Visualize a project management tool. The project management tool stores information and presents several views (abstractions) to the user. Typical views are GANTT and PERT charts. In a similar manner, a new type of OOSE tool could present information in the user diagrammatic format (e.g. Process flow charts) as well as in developer format (UML or whatever development methodology becomes the prevalent paradigm in the future).

Thus the next generation of SE tools may have the capability to describe user needs in ways that are both OO and better than the current generation of text-mode based requirements tools. In addition, once an OOSE/OOSWE language becomes the language of design, other currently troublesome issues may become moot. For example, the design of interfaces between different engineering tools will be easier. Interface designers will just face the relatively simple problem of transferring the content of the language rather than the thorny problem of "sharing meaning" (Harris 2000).

TRUE OBJECT-ORIENTED SYSTEM ENGINEERING

SE is inherently OO and needs only a slight enhancement to seamlessly interface with FCD based OOSWE. This seamless interface must be implemented by simplifying the current process, not by adding a layer of complexity between systems and SWE. The proposal for simplification herein is to make a few changes to the way SE is performed, namely:

1. Use the concept of inheritance so that requirements" are not missed by design engineers lacking experience and domain expertise.
2. Partition the system for optimal minimally coupled system level component designs. This will mean that system and software engineers will have to work together in the up-front stages of the SDLC before the partitioning into subsystems takes place. This no different to the system engineer working with other technical engineering specialists such as reliability, manufacturing and thermal engineers.
3. Stop referring to functional and non-functional requirements and start referring to properties needed (by customers) and properties provided (by components in existence or to be built).

CONCLUSIONS

SE is indeed inherently OO. The major differences in the OO approach between SE and SWE seem to be that textbook SE looks at alternatives and then selects the optimal one based on a set of evaluation criteria. Thus SE seems to operate at the UML meta-model level. SWE on the other hand, tends to pick one model or solution early in the design process and run with it.

Using an OO approach and the appropriate next generation tools to capture and track properties should result in more systems being built correctly the first time avoiding costly rework after delivery (Kasser 2000).

SE is all about ensuring that all the properties of the system as delivered (system capability) are at least equal to the properties of the system needed (system needed). Thus it is requirements free.

REFERENCES

Blanchard and Fabrycky, *SE and Analysis*, Prentice Hall, 1981.

C4ISR Architecture Framework, Version 2.0, US Department of Defense, December, 1997.

Capers Jones, *Patterns of Software Systems Failure and Success,* International Thomson Computer Press, 1996.

Chang, C.K., Hua, S.,"A New Approach to Module-Oriented Design of OO Software*", 18th International Computer Software and Applications Conference, (COMPSAC94),* Los Alamitos, CA., 1994.

Chang, C.K., Cleland-Huang, J., Hua, S., Kuntzmann-Combelles, A., "Function-Class Decomposition", A Hybrid Software Engineering Method", IEEE Computer, December 2001, pp 87-93.

CHAOS, "The CHAOS Report", The Standish Group, 1995, http://www.pm2go.com/sample_research/chaos_1994_1.asp, last accessed January 21, 2002.

Checkland, P.*, Systems Thinking Systems Practice*, Wiley, 1981.

Eisner, H., *Computer Aided SE*, Prentice Hall, 1988.

Emmerich, W., *Engineering Distributed Objects*, John Wiley & Sons, 2000.

Gabb, A., "Front-end Operational Concepts - Starting from the Top", *The 11th Annual Symposium of*

the INCOSE, Melbourne Australia, 2001.

Gabb, A., Caple, G., Haines, D., Lamont, D., Davies, P., Hall, A., van Gaasbeek, J., Eppig, S., Jones, D., Vietinghoff, B., "Requirements Categorization", *The 11th Annual Symposium of the INCOSE*, Melbourne Australia, 2001.

Gabbar, H.A., Shimada, Y., Sukuzi, K, "Object Interface System Using a Plan Object-Oriented Model", Systems Engineering, Volume 4 Number 3, 2001.

Gibbons, P. J., "A Case Study in the Application of UML and OOA/D in an Information Management Program", *The 11th Annual Symposium of the INCOSE*, Melbourne Australia, 2001.

Glinz, M., "Problems and Deficiencies of UML as a Requirements Specification Language, *Proceedings of the 10th International Conference on Software Specification and Design*, San Diego, CA, 2000.

Hambleton, K.G., Systems Engineering Principles, Lecture Notes, University College London, 1999.

Harris, David D,: "Supporting Human Communication in Network-based Systems Engineering", *Proceedings 2nd European Systems Engineering Conference*, Munich, Germany, September 13-15, 2000, pp 221-226.

Hatley, D. J., Pirbai, I. A., *Strategies for Real-Time System Specification*, Dorset House, 1994.

Hooks, I., "Writing Good Requirements", *Proceedings of the 3rd NCOSE International Symposium, 1993,* available at http://www.incose.org/rwg/writing.html, last accessed November 1, 2001.

Jacobs, S., "Introducing Measurable Quality Requirements: A Case Study", *IEEE International Symposium on Requirements Engineering*, Limerick, Ireland, 1999.

Jorgensen, R., "The Oxymoron of Use Case Requirements", *INSITE, Volume 4 Issue 2*, July 2001.

Kasser, J.E., Schermerhorn, R., "Gaining the Competitive Edge through Effective Systems Engineering", *the 4th Annual International Symposium of the National Council on Systems Engineering (NCOSE)*, San Jose, CA., 1994.

Kasser, J.E., "Enhancing the Role of Test and Evaluation in the Acquisition Process to Increase the Probability of the Delivery of Equipment that Meets the Needs of the Users", *The Systems Engineering, Test and Evaluation Conference (SETE 2000)*, Brisbane, Australia, 2000.

Kasser, J.E., "A Web Based Asynchronous Virtual Conference: A Case Study", *The INCOSE - Mid-Atlantic Regional Conference*, Reston, VA, 2000a.

Kasser, J.E., "A Prototype Tool for Improving the Wording of Requirements", *The 12th Annual Symposium of the INCOSE*, Las Vegas, NV, 2002.

Kline, S.J., *Conceptual Foundations for Multi-Disciplinary Thinking*, Stanford University Press, 1995.

Kobyn, C., "UML 2001: A Standardization Odessy", Communications of the ACM, October 1999, Vol 42 No 10, Available at http://www.omg.org/attachments/pdf/UML_2001_CACM_Oct99_p29-Kobryn.pdf, last accessed November 21, 2001.

Lagakos, V., Kaisar, E. I., Austin, M. "Object Modeling for the Management of Narrow Passageways in Transportation Systems", *The 11th Annual Symposium of the INCOSE*, Melbourne Australia, 2001.

Maimonides M., (1135-1204), *The Guide for the Perplexed*, translated by Friedlander M, Dover Publications, 2nd Edition, 1956, PP 69-70.

Menzes, T., Easterbrook, S., Nuseibeh, B., Waugh, S., "An empirical investigation of multiple viewpoints reasoning in requirements engineering", *IEEE International Symposium on Requirements Engineering*, Limerick, Ireland, 1999.

OASIG, The performance of information technology and the role of human and organizational factors. Report to the Economic and Social Research Council, UK, 1996, available at http://www.shef.uc.uk/~iwp/publications/reports/itperf.html, last accessed January 16, 2002.

Ogren, I., Comment on the *use case* Requirements Oxymoron, *INSITE Volume 4, Issue 3*, October 2001.

Somerville, I., *SWE*, 5th edition, Addisson-Wesley, 1995.

Sutcliffe, A., Galliers, J., Minocha, S., "Human Errors and System Requirements", *IEEE International Symposium on Requirements Engineering*, Limerick, Ireland, 1999.

UML, OMG Unified Modeling Language Specification Version 1.3. OMG document ad/99-06-08.

Van Gaasbeek, J. R., Martin, J. N., "Getting to Requirements: The W5H Challenge", *The 11th Annual Symposium of the INCOSE*, Melbourne Australia, 2001.

Van Vliet, Hans, *OO Analysis and Design*, 2nd Edition, John Wiley & Sons Ltd., 2000

BIOGRAPHY

Joseph Kasser has been a practicing systems engineer for 30 years. He is the author of "*Applying Total Quality Management to Systems Engineering*" published by Artech House. Dr. Kasser is a DSTO Associate Research Professor at the University of South Australia (UniSA). He performs research into im-

proving the acquisition process. Prior to taking up his position at UniSA, he was a Director of Information and Technical Studies at the Graduate School of Management and Technology at University of Maryland University College. There, he developed and was responsible for the Master of Software Engineering (MSWE) degree and the Software Development Management (SDM) track of the Master of Science in Computer Systems Management (CSMN) degree. He is a recipient of NASA's Manned Space Flight Awareness Award for quality and technical excellence (Silver Snoopy), for performing and directing systems engineering. Dr. Kasser also teaches SE and SWE in the classroom and via distance education.