A Design Quality Learning Unit in OO Modeling Bridging the Engineer and the Artist

Leslie J. Waguespack
lwaguespack@bentley.edu
Computer Information Systems Department
Bentley University
Waltham, Massachusetts 02452, USA

Abstract

Recent IS curriculum guidelines compress software development pedagogy into smaller and smaller pockets of course syllabi. Where undergraduate IS students once may have practiced modeling in analysis, design, and implementation across six or more courses in a curriculum using a variety of languages and tools they commonly now experience modeling in four or fewer courses in at most a couple of paradigms. And in most of these courses their modeling decisions focus on acceptable syntax rather than principles representing and communicating concepts of quality in information systems. Where learning design quality may once have been an osmotic side effect of development practice it must now be a conscious goal in pedagogy if it is to be taught at all. This paper presents a learning unit that teaches design quality in object-oriented models. The focus on object-oriented models allows the learning to permeate analysis, design, and implementation enriching pedagogy across the systems development life cycle. The quality perspective presented is more expansive than that usually found in software engineering, the traditional "objective" notion of metrics, and integrates aspects of aesthetics, the more subjective phenomena of satisfaction. This learning unit is intended as an adaptable framework to be tailored to the coursework and the overall objectives of specific IS programs.

Keywords: design quality, design, OO modeling, IS discipline, IS curricula, IS pedagogy

1. INTRODUCTION

Over the past decade computing curricula have been repartitioned with the permeation of computing across disciplines and society. (Shackelford, Cross, Davies, Impagliazzo, Kamali, LeBlanc, Lunt, McGettrick, Sloan & Topi, 2005) There are now 5 major computing curriculum guidelines that subdivide computing. (Soldan, Hughes, Impagliazzo, McGettrick, Nelson, Srimani & Theys 2004, Cassel, Clements, Davies, Guzdial, McCauley, McGettrick, Sloan, Snyder, Tymann & Weide, 2008, Diaz-Herrara & Hilburn, 2004, Lunt, Ekstrom, Gorka, Hislop, Kamali, Lawson, LeBlanc, Miller & Reichgelt, 2008, Topi, Valacich, Wright, Kaiser, Nunamaker, Sipior & de Vreede, 2010) The co-location of IS curricula in schools of business further exacerbates the pressure on pedagogy as accreditation bodies further constrain the scope of coursework by compressing systems development into smaller and smaller pockets of course syllabi. (AACSB 2010, EQUIS 2010) Where undergraduate IS students once may have practiced modeling in analysis, design, and implementation across six or more courses in a program using a variety of languages and tools they commonly now experience modeling in four or fewer courses in at most a couple of paradigms. (Waguespack 2011) And in most of these courses their modeling decisions focus on acceptable syntax rather than principles representing and communicating concepts of quality in information systems. Where learning design quality may once have been an osmotic side effect of development practice it must now be a conscious goal in pedagogy if it is to be taught at all.

At the same time industry and academia persist in their lament over the paucity of focus on quality in system design first sounded more than four decades ago (Dijkstra, 1968) and echoing consistently since as in (Denning, 2004, Brooks 1995, 2010, Beck, Beedle, van Bennekum, Cockburn, Cunningham, Fowler, Grenning, Highsmith, Hunt, Jeffries, Kern, marick, Martin, Mellor, Schwaber, Sutherland, & Thomas 2010)

This paper presents a learning unit that teaches design quality within the object-oriented paradigm. The focus on OO models allows the learning to permeate analysis, design, and implementation enriching pedagogy across the systems development life cycle. We amplify a traditional "objective" notion of systems quality (i.e. metrics usually found in software engineering) by integrating the more subjective phenomena of satisfaction, aesthetics. This learning unit is adaptable to the coursework and objectives of specific IS programs. The paper presents: a brief overview of design quality, properties to assess design choices, the object-oriented ontology; and a discussion of how each of the design choice properties express quality through the use of object-oriented modeling constructs. Finally, there is a description of how the learning unit has been integrated in object-modeling syllabi with a comment on its efficacy.

2. WHAT IS DESIGN QUALITY?

Quality is an elusive concept, shifting and morphing on a supposed boundary between science and art: objective, engineering characteristics versus subjective, aesthetic observer or stakeholder experience. International standards of quality reflect the challenge of defining quality by offering a variety of perspectives (as gathered here by Hoyle 2009):

- A degree of excellence (Oxford English Dictionary)
- Freedom from deficiencies or defects (Juran 2009)
- Conformity to requirements (Crosby 1979)
- Fitness for use (Juran 2009)
- Fitness for purpose (Sales and Supply of Goods Act 1994)
- The degree to which the inherent characteristics fulfill requirements (ISO 9000:2005)
- Sustained satisfaction (Deming 1993)

(Waguespack 2010b) asserts that the quality of systems revolves around two primary concepts: efficiency and effectiveness defined as follows (New Oxford American Dictionary):

Efficiency [noun]- the ratio of the useful work performed […] in a process to the total energy [effort] expended

Effectiveness [noun]- successful in producing a desired or intended result

These two concepts appear primarily quantitative and therefore objective. In and of themselves they may well be. Portraying efficiency using a convenient interpretation of "work" and "effort" is genuinely objective. "How many" or "how much" or "how often" often depicts efficiency. But, when we ask "Is it enough?" apparent objectivity fades away.

Likewise, the supposed objectivity of "effectiveness" relies upon the tenuous phrase, "desired or intended result" defined as

Intend [noun]- have (a course of action) as one's purpose or objective; plan

Effectiveness (like efficiency) is a correspondence between a system and its stakeholders' intentions. Assessing effectiveness depends on comparing "what is" to "what is intended." While the former may be expressed quantitatively the latter presents challenges: clarity of conception, mode of representation, scope of contextual orientation, and fidelity of communication to name but a few. Indeed the notion of effectiveness is complicated when we contemplate identifying and quantifying the stakeholder(s) intentions objectively.

The indefiniteness or imprecision that characterizes stakeholder intention(s) is generally not a concern if an observer is asked to assess the beauty of something – an assessment generally conceded to be subjective. A detailed or even explicit intention is not expected in assessing beauty – beauty is most often perceived as an experience of observation rather than a system analysis. Most people commonly accept beauty as subjective and exempt from specific justification or explanation – "Beauty is in the eye of the beholder." and "You'll know it [beauty] when you see it." This absence of or difficulty in forming a quantitative justification of beauty is often the basis for categorizing artifacts or processes as products of art rather than of engineering. And therein lies the presumption that the aspects of design quality that we label objective and those we label subjective are somehow dichotomous. They in fact teeter between objectivity and

subjectivity depending on the degree of granularity that observers choose to employ in inspecting not only the artifact but also their own disposition toward satisfaction relative to it.

3. AN ARCHITECTURAL INTERPRETATION OF QUALITY DESIGN

We will never be able to absolutely define design quality because of the relativistic nature of satisfaction in the observer experience. But, our students must still face design choices. So, as IS educators we must provide a framework for them to develop and refine their individual perceptions and understanding of systems quality. The taxonomy of design choice evaluation proposed in Waguespack (2008, 2010b), the 15 *choice properties*, is just such a framework. (See Appendix A.) Choice properties derive from Christopher Alexander's writings on design quality in physical architecture. (Alexander 2002)

Choice properties address the process of building, the resulting structure, and the behavior of systems as cultural artifacts. Every design decision, choice, contributes to the aggregate observer experience: either positively or negatively. Each choice exhibits the 15 properties with varying strengths or influence that impact the resulting observer satisfaction. The confluence of property strength results from the coincidence of the designer's choice with the collective intention of the stakeholders. The combination of all choices with their respective property strengths results in the overall, perceived design quality. Many of the properties are design characteristics long recognized in software engineering (i.e. modularization, encapsulation, cohesion, etc.). But several reach beyond engineering to explain aesthetics, the art (i.e. correctness, transparency, user friendliness, elegance, etc.). An example of the effectiveness of choice properties in explaining the design quality of production systems is reported in (Waguespack, Schiano & Yates 2010a).

4. THE ONTOLOGY OF THE OBJECT-ORIENTED PARADIGM

Illustrating design decisions in the object-oriented paradigm can be a challenge. The idiosyncrasies of OO programming syntax often obscure the intention and/or the result of a design decision. For that reason the learning unit presented here uses a paradigm description independent of programming language, the object-oriented ontology, found in (Waguespack 2009) and excerpted in Appendix B. The graphical outline of the ontology is Figure 1 below.

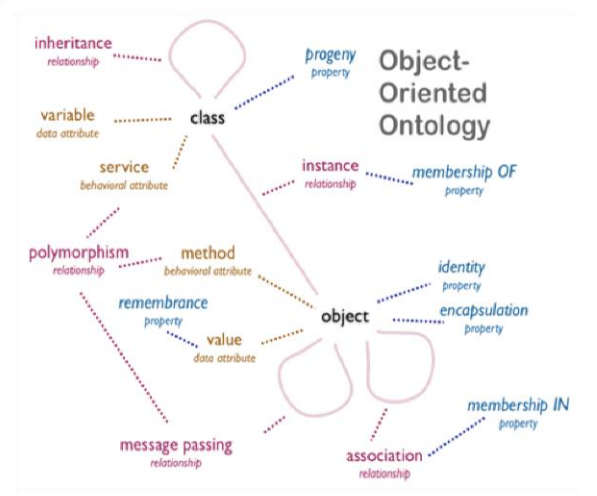


Figure 1 – Object-Oriented Ontology

The ontology captures the elements of the object-oriented paradigm eschewing the obfuscation that usually occurs with programming language syntax examples. At the same time an experienced IS teacher can readily translate the ontological elements into a relevant programming dialect.

5. CRAFTING OBJECT-ORIENTED MODELING CHOICES THAT STRENGTHEN PROPERTIES OF DESIGN QUALITY

This section, the heart of the learning unit, enumerates the 15 choice properties as defined in Waguespack (2010b) illustrating how modeling choices in the object-oriented ontology can express design quality. In this space-limited discussion one choice property often references another reflecting the confluent nature of the design quality properties as Alexander defines them in physical architecture. (Alexander 2002)

Stepwise Refinement (as the name implies) is an approach to elaboration that presumes a problem should be addressed in stages. The stages may represent degrees of detail or an expanding problem scope. (Birrell and Ould 1988) In either case quality evidence of *stepwise refinement* is demonstrated by the cogent and complete representation of a design element at whatever level of detail or scope is set at each stage. To achieve this representation the modeling paradigm must support abstraction that allows generalization of the scope of interest and then the elaboration of that scope from one stage to the next.

The class concept in OO provides this capability. Through the inheritance relationship a class can

represent the more abstract, general character of a model feature while expressing all the information and behavior needed at that level of abstraction: 1) what responsibilities the objects of this class fulfill, 2) what information they manage, and 3) what services this class's objects provide the rest of the model. As the modeling stages progress greater specialization is achieved with child classes that redefine abstract behaviors: by adding data and/or behavioral attributes germane only at a lower level of abstraction, or by defining collaborations to support this class's responsibilities. *Stepwise Refinement* can mimic the concept of "need-to-know." Only that detail required to "understand" the system at that abstraction level need be revealed or perhaps is not even chosen until the need arises. When the need does arise the detail may be added within the genealogy of the class preserving the *cohesion* of a class's defined functional responsibility at the higher abstraction levels.

As an example, consider a class that defines items stored in an inventory. At the most general level the most important functional detail is the entry and removal of items. As refinement progresses simple entry and removal may be augmented by including item re-order and supplier interaction both concealed from the inventory item's client who sees only entry and removal. The supplier interaction details are *encapsulated* within the inventory item's responsibilities retaining the *cohesion* of the class's purpose (its *identity)*. And the description of the inventory item exhibits *correctness* at either level of detail with and without the supplier interaction elaboration.

Cohesion is a quality property reflecting a consistent responsibility distribution in a field of system components. (Zuse 1997) Since every object "expects" the objects around it to fulfill their responsibilities to contribute to the whole model, each object is in itself free to be single-minded in its focus on its own purpose. This is the result of well-chosen classes. This independent sufficiency accentuates the divisibility of function in terms of each object's individual purpose, its *identity*, and the clarity with which its purpose is exposed to the rest of the community of objects in the system. The single-mindedness that results also increases the feasibility of object interaction rearrangement enabling an overall change in system function while almost every class's individual purpose remains fixed. The independent sufficiency of each object's inner workings couples with the system-wide interdependency of object cooperation to promote a texture exhibiting a sense of system connectedness, *elegance.*

Encapsulation is a design quality reflected directly in the nature of the object-oriented ontology as objects *encapsulate* both their data and behavioral attributes. *Encapsulation* clearly delineates who is allowed to manipulate system information and who is not. Object data and behavior are only accessible (invoke-able) via the published services defined for each object by its class. When sustained as a discipline this boundary universally designates the object as the finest granule of *modularization*. (Scott 2006) This principle eliminates the possibility of "side effects" where system state changes occur in any manner other than the "contractual" prescription defined in the object's service interface. The isolation of the inside of the object from the outside allows both to evolve without servitude to the implementation of the other (e.g. pursuing efficiency) as an object is obligated only through the published responsibilities in its class's services.

Extensibility is the property of design quality most important in pursuing systems with sustainability essential to cost of ownership economy. This is the vehicle for seamless unfolding in system evolution. Extensibility juxtaposes the potential for new functionality with the effort required to achieve it. (van Vliet 2008). In the object-oriented paradigm class plays the pivotal role by empowering instance and inheritance relationships.

Multiplicity is achieved through instance propagation, *progeny*. Each instance is completely interoperable in any combination with its sibling objects as well as acting as an instance of any ancestor class. Interchangeability both enables and reinforces *modularization*.

Evolution or unfolding is accomplished as class definitions are refined and specialized in their child classes – the relationship called *inheritance*. When a child class extends the scope of the data and behavioral attributes of its parent it honors the pattern set out in the parent without contradiction. *Polymorphism* compensates (through dynamic binding) for any overridden methods. This extension proceeds without any impairment of *correctness* because the interfaces defined in the parent class must be supported in each child class. The parent to child unfolding specializing structure and behavior results in an unbroken thread that binds each class to its ancestry and projects an *identity* down through the generations of class.

Modularization along with *cohesion* expresses "divide and conquer" problem solving augmented by the flexibility of configuring and reconfiguring objects as cooperating agents. *Modularization* also supports *scale* permitting the composition of subsystems of varying scope that hold details in abeyance until they require focus. (Baldwin and Clark 2000) Enlightened module design exposes the solution structure envisioned by the modeler and publishes intentions for further extension by separation of concerns and isolation of *accidents of implementation*. (Brooks 1987) The OO paradigm provides ample facility for defining modules of any size and scope while aggregating and/or nesting their interfaces through deliberate information hiding. The granularity enabled through *modularization* may be applied to facilitate the modeler's formulation of structure as well as the perspective to aid stakeholder recognition and understanding.

Correctness in software engineering is often narrowly defined as computing the desired function. (Pollack 1982) Thriving Systems Theory frames this property upon two outcomes: 1) validation, the clarity and fidelity of the represented understanding of system characteristics, and 2) verification, the completeness and effectiveness of model feature testing both individually and in composition.

Validation depends on the fidelity of the unfolding process; that through the stages of *stepwise refinement* the "essence" of system characteristics are brought forward maintaining their integrity. (Brooks 1987) *Modularization* aids in cataloging and focusing on individual essential characteristics. *Correctness* is the only choice property that directly supports itself! *Correctness* must be a priority at each stage as experience shows that *correctness* shortcomings grow more and more expensive to rehabilitate as evolution progresses – notice "rehabilitate," to restore to normal *life*.

Verification depends on the effective testability of each choice to certify it as "consistent with stakeholder understanding." *Modularization* enables the verification of individual choices or modules. Then relying on the *correctness* inside modules verification can turn to the certification of behaviors resulting from *composition of function*. Experience often leads to dependable *patterns* of classes or modules applicable or adaptable to recurring modeling tasks. Verification in these situations can focus on known areas of fragility/risk limiting the effort required to reach a desired confidence level of *reliability*.

Transparency is evident structure, revealing how things fit and work together. (Kaisler 2005) In the OO ontology "fit together" and "work together" are defined by the structural and behavioral relationships. Individual objects may represent clearly delineated and encapsulated choices, but their cooperation is defined by relationships.

Inheritance explains the structural relationship of classes through the propagation of data and behavioral attributes. Inheritance not only propagates attributes, but also enables a class hierarchy's capacity for exhibiting similarity and difference between parent and child classes. That which is similar (in fact identical) inherited by the child class is assumed and becomes in effect familiar – requiring no reiteration. This "folding" of that which is not changed avoids clutter in the child class description, but may be readily reviewed in the parent.

The behavioral relationships of association, message passing, and polymorphism explain the predictable *patterns* of communication and action. Association uses the property of *identity* to designate membership, ownership, and accessibility among objects. Message passing provides the mechanism for cooperating action between objects providing a disciplined conduit through the encapsulating boundary of objects by using services to convey intention, information, and reaction. Polymorphism allows the abstraction of intention by using the same service name to evoke distinct behaviors from objects of different classes. The identical service names in classes with different methods directly realize the metaphorical abstraction of object behavior where at one level of abstraction the behaviors are the same and at a more detailed level of abstraction their behaviors are distinct.

Composition of Function - As a fundamental tool for managing complexity humans regularly attempt to decompose problems, issues, or tasks into parts that either in themselves are sufficiently simple to permit direct solution or can through recursion be subdivided successively until they become sufficiently simple. This is a defining aspect of *modularization*. When the conception of a part also anticipates reuse then the part takes on a larger significance. The combination of specifying a choice consistent with the essence of system characteristics and then designing the choice as an interchangeable component in multiple super-ordinate choices is a step toward *elegance*. Reusable choices represent an understanding of the essence of the system at a deeper level than an individual application. They

represent awareness of the intention, perhaps even the philosophy of the system domain.

Composition of function as a property of design quality is realized in model features that facilitate the extension or retargeting of the model in the future. It is the capacity to combine simple functions to build more complicated ones (Meyer 1988). The retargeting capability may be provided directly to the users of the system in the form of a *programmable* interface. A choice achieving the principle of *composition of function* is marked not only by the function it initially provides the user, but also by the functionality it anticipates and supports even (perhaps) *before* the stakeholders realize the need for the capability.

Identity is at the root of recognition and is another property of design quality not usually defined in software engineering. In the physical world *identity* is literal based upon direct sensorimotor experience: by sight or touch and in some cases by sound or smell – a human experience of the "real" world. In the object-oriented paradigm *identity* is an object property. (Khoshafian and Copeland 1986) Existence is sufficient for object identification.

In other paradigms identification is achieved through possessed characteristics (attributes) that contribute to distinct recognition by a process of intersecting categorizations or the introduction of an artificial characteristic whose sole purpose is to support discrimination. Aside from the fact that these approaches to identification require some overhead (either mental or computational) they are simply not natural to humans. Humans perceive objects as possessing characteristics rather than characteristics defining objects. The former begins with certain uniqueness and progresses toward explanation while the latter begins with uncertainty and attempts to deduce uniqueness.

Characteristics are not unimportant. Classification is essential in most human problem solving activities. And recognition is virtually always accelerated by the discrimination that categorizing characteristics (attributes) provide. And most importantly in the absence of physical experience categorization through characteristics is the only choice. Class structure and the instance relationship are vital to *identity* – an object belongs to "this" class and not to "another." Described both by what an object "knows" (data attributes) and what it "knows how to do" (behavioral attributes) classes form a categorization cornerstone of the object-oriented ontology. But to model both the static and dynamic dimensions of reality (association and message passing) each object must be uniquely distinguishable.

Scale's effect on design quality is reflected in common idioms: "You can't see the forest for the trees!" and "Let's get a view from 10,000 feet." They reflect the importance of context in recognition and decision-making. *Scale* captures the modeling imperative that all choices must be kept in perspective because it is not sufficient to consider a choice only in the microcosm of itself, as it must also participate in the connectedness of the whole. By achieving scale, a system designer provides differing granularities of comprehensibility to suit the requirements of a variety of observers (Waguespack 2010).

The relationships provided in the object-oriented paradigm (association, inheritance, instance, message passing, and even polymorphism) provide ample means for designing collections of cooperating choices that are nested, intersect, or partition the full *field* of functionality essential to the model. These may be called variously subsystems, modules, or sub-modules. In those cases where the actual structure of a collection must be rendered obscure, classes and objects can be devised to serve as facades or agents to "keep up appearances." Coupled with *stepwise refinement,* as it is, *scale* is used to focus modeler and stakeholder attention to achieve the contextual understanding needed to address constituent concerns within the whole.

User Friendliness is another property of design quality more often considered aesthetic. It is a combination of: ease of learning; high speed of user task performance; low user error rate; subjective user satisfaction; and, user retention over time (Shneiderman 1992). Its impact may be easiest to consider in its absence. A modeling choice that is "unfriendly" to stakeholders is confusing, hard to comprehend, unwieldy, and perhaps worst of all, of indeterminate *correctness*. That which defies understanding cannot be determined to be correct. Satisfaction is cumulative. The sensitivity to the stakeholders' conceptions of the essence of the system to be modeled is key to the stakeholders' sense of comfort, familiarity, and expectation.

The object-oriented paradigm excels in its facility to represent systemst preserves the stakeholders' ability to recognize "their" system. Authoring object-oriented models whose elements correspond almost one-to-one with the real-world concepts and entities results in intrinsically better stakeholder understanding and interaction. The casting of "objects" in the models that have direct counterparts in the stakeholders' experience exhibits a

fundamentally friendly quality. It respects the stakeholders' perceptions and it welcomes them into the processes of verification and validation that are intrinsic to *correctness*. The unified structure of "what an object knows" and "what an object knows how to do" correlates so naturally with observers of business models or process models that the natural clarity in that communication improves understanding and avoids mistakes in understanding, communication, or implementation.

And in a serendipitous quirk of language (or a profound emergence of the deep meaning of metaphors) Alexander's term from which the principle here, *user friendliness*, is derived is *roughness*. (Alexander 2002) Something has to have a certain degree of *roughness* if one is to be able to effectively grasp it!

Patterns describe versatile templates to solve particular problems in many different situations (Gamma et al. 1995). All entities in the object-oriented paradigm propagate from classes, predefined templates, or "cookie cutters." This protocol organizes what otherwise would be a bewildering multiplicity of individual computational entities to consider. It becomes less complicated in the understanding that the potential of any number of objects boils down to understanding the class(s) of which they are instances. Each instance mimics perfectly the form and function of every other of its siblings, members of that class. Class hierarchies, generations of parent-child class definitions, defining "nearly the same" and "different in specific ways" relationships significantly lessen the apparent complexity that considering only individual entities entails. Class hierarchies define the path of *unfolding* for all to see – a depiction of the analysis, solution, and design philosophies at work.

Patterns is the property of design quality that channels change (unfolding). A pattern foreshadows where and how change will need to be accounted for. Patterns of the form popularized in (Coplein, 1995) document commonly encountered design questions offering carefully considered advice and cautions. Their patterns are paradigm and modeling language independent. However, it is not surprising that many examples using *patterns* are presented in OO dialects. The reason is simple. The integration of instance, inheritance, message passing, and polymorphism relationships is an ideal toolset for expressing *patterns* with a balance of prescription and adaptability – a balance not as conveniently achieved in dialects based on pre-object-oriented paradigms.

Programmability in software engineering is often considered a feature rather than a property of design quality – the capability within hardware and software to change; to accept a new set of instructions that alter its behavior (Birrell and Ould 1988). It is closely allied with *extensibility*and addresses the need for models to welcome the future. What largely separates information systems from other human-made mechanisms is the degree of adaptability that they offer to deal gracefully with change. Unlike most appliances that support a very narrow range of use (albeit with great *reliability*), contemporary information systems are expected to provide not only amplification of effort as in computation, but also amplification of opportunity in terms of different approaches to business or organizational questions. Contemporary information systems are expected to demonstrate that they can reliably accommodate change. As with *extensibility,* successful accommodation of change relies on an understanding of the fundamental options governing the structure and behavior within a particular domain. The OO ontology offers powerful tools (structural and behavioral relationships, e.g. inheritance and polymorphism) to service the elements of change without fracturing a skeletal foundation of base classes characterizing the domain.

What sets *programmability* apart from *extensibility* is a facility that permits altering the systems behavior without having to reconstruct choices – that is to say that the system's behavior can be sensitive to the context determined by a "user" in "real time." "Real time" is relative to the "user's" role (e.g. developer or end user, etc.). This versatility is not accidental but architectural. Choices may provide an interface language for end users that permits selections of system actions to meet an immediate "real-time" need – an interface as simple as a light switch or as complex as a natural language.

Reliability is a property of design quality more often associated with implementation than design. It is the assurance that a product will perform its intended function for the required duration within a given environment (Pham 2000). Objects facilitate modularized testing and quality assurance. A certified class produces certified objects (which is not to say that certification is easy or inexpensive). As long as classes are protected from dynamic modification in deployment there is no need to be concerned with the inner workings of their objects. As long as objects are truly *encapsulated* they conform to the intention of their class. In development

testing proceeds incrementally as new classes are added or rearranged in their collaboration. Once deployed testing is relegated to their interactions rather than their definition. Testing is compartmentalized and does not explode exponentially when additional classes or functionality within a class is added.

Reliability in design reflects an austerity that confines design elements to the essentials of the stakeholder's intentions. When design or implementation decisions involve additional constructs due to technology or compatibility, these *accidents of implementation* must be clearly delineated so as not to imply that they are essence rather than accident. This clear distinction will protect future system evolution from mistaking accidental "baggage" as stakeholder intentions.

Elegance is perhaps the epitome of subjective quality assessment that clearly sets choice properties of design quality apart from traditional software engineering metrics. "Pleasing grace and style in appearance or manner," that's how the dictionary expresses the meaning of "elegance". (Oxford English Dictionary)

> "A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away." (Raymond 1996)

Models composed of choices that are consistent, clear, concise, coherent, cogent, and transparently correct exude *elegance* and nurture cooperation, constructive criticism and stakeholder community confidence. These are models that confess to their own shortcomings because their clarity obscures nothing, even omissions. These are models that satisfy stakeholders. They appear "intuitively obvious." The clarity of their composite structure is so self-evident that they seem "simple." The use of the OO paradigm to construct a collection of "building blocks" in the form of a class library to encapsulate architectural design decisions facilitates this impression of what is "intuitively obvious." Using well-conceived library elements becomes so second nature, so natural, that the builder perceives the blocks as the natural primitives of construction rather than constructed artifacts.

Elegance largely proceeds from the efficient and effective representation of *essential* system characteristics along with those features emerging out of design decisions, *accidents of implementation*, that are laid out with equal clarity for separate consideration. This is the *field effect* of the beneficial, integrated, mutual support of strong choices described in Thriving Systems Theory. (Waguespack 2010b)

6. INTEGRATING THE DESIGN QUALITY LEARNING UNIT IN AN OBJECT-ORIENTED MODELING SYLLABUS

For the past six semesters the design quality learning unit presented here is woven into two object-oriented modeling syllabi: 1) undergraduate systems analysis and design and 2) masters level object-oriented systems engineering. The unit content appears throughout the pedagogy of modeling using UML-2 syntax.

After initially presenting the object-oriented paradigm using the ontology to establish its vocabulary (see Appendix B), we present use case, class, and sequence diagramming establishing the syntax and the expression of semantics in UML-2. During this UML presentation we repeatedly allude to the design quality properties through the syntax. Small student groups and then individuals conduct a series of modeling exercises based on requirement narratives establishing the students' grasp of UML syntax. On that foundation the explanation of design quality, the enumeration of the fifteen properties, and the corresponding application of OO ontology elements to strengthen the properties precede a final individual course modeling project. The design quality discussion provides a quality vocabulary for one-on-one consultations between teacher and student as each develops the object-model of their final project. In this one-on-one context each student's specific design decisions are discussed and evaluated in relationship to the design quality properties, an opportunity for individualized, reinforced learning and/or suggested improvements.

The deeper subtleties of design quality present a challenge for some students particularly in a compressed format. The "light doesn't go on" right away for all students. However, the integration of the ontology and design quality property based vocabulary establishes a touchstone that returning students report helps them "to name" the "quality elements" they rediscover in succeeding coursework and professional practice.

In your own curricular situation the distribution of learning unit elements may span more than one course (some addressed in OO programming, requirements engineering, or database design, etc.), be rearranged to suit

your modeling tools, or be adjusted to your course sequencing with context-appropriate examples. Regardless, the learning unit components are flexible and robust enough to suit various specific program needs.

7. ACKNOWLEDGEMENTS

Thanks to helpful referees. Special thanks are due my colleagues David Yates and Bill Schiano at Bentley University for their insightful discussions and comments on these ideas.

8. REFERENCES

AACSB (2010). Eligibility Procedures and Accreditation Standard for Business Accreditation. Retrieved July 16, 2010 from http://www.aacsb.edu/accreditation/AAACSB-STANDARDS-2010.pdf

Alexander C, (2002). The Nature of Order An Essay on the Art of Building and the Nature of the Universe: Book I - The Phenomenon of Life, Berkeley, California: The Center for Environmental Structure, p. 119

Baldwin, C. Y., and Clark, K. B. (2000). Design Rules, Volume 1: The Power of Modularity. The MIT Press, Cambridge, MA.

Beck K., Beedle M., van Bennekum A., Cockburn A., Cunningham W., Fowler M., Grenning J., Highsmith J., Hunt A., Jeffries R., Kern J., Marick B., Martin R.C., Mellor S., Schwaber K., Sutherland J., & Thomas D. (2010). *Manifesto for Agile Software Development*. Retrieved July 12, 2010 from agilemanifesto.org

Birrell, N. D., and Ould, M. A. (1988). A Practical Handbook for Software Development. Cambridge University Press, Cambridge, UK.

Brooks F. P. (1987), "No Silver Bullet: Essence and Accidents of Software Engineering," Computer, Vol. 20, No. 4, pp 10-19.

Brooks, F. P. (1995). The Mythical Man-Month: Essays on Software Engineering (2ed). Addison-Wesley, Boston, MA.

Brooks, F. P. (2010). The Design of Design: Essays from as Computer Scientist. Addison-Wesley, Pearson Education, Inc., Boston, MA.

Cassel L., Clements A., Davies G., Guzdial M., McCauley R., McGettrick A., Sloan B., Snyder L, Tymann P., & Weide B.W., (2008). Computer Science Curriculum 2008 An Interim Revision of CS2001. Association of Computing Machinery (ACM), & IEEE Computing Society (IEEE-CS)

Coplien J and Schmidt D (Eds) (1995). Pattern Languages of Program Design, Addison-Wesley, Reading, MA, USA

Crosby,P. B., (1979) Quality is Free, McGraw-Hill, New York, NY, USA.

Dijkstra, E. (1968). "GOTO Statement Considered Harmful." *Communications of the ACM*, 11(3), 147-148

Diaz-Herrara, J.L., & Hilburn, Thomas B. (eds.) (2004). Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering, IEEE Computing Society (IEEE-CS), Association of Computing Machinery (ACM)

Deming, W. E. (1993), The New Economics for Industry, Government, Education (2ed), Cambridge Press: MIT, Cambridge, MA, USA

Denning, P. J. (2004). "The Great Principles of Computing," *Ubiquity*, 4(48), 4–10

EQUIS (2010). *EQUIS Standards and Criteria*. Retrieved July 16, 2010 from http://www.efmd.org/attachments/tmpl_1_art_041027xvpa_att_080404qois.pdf

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA.

Hoyle, D. (2009). ISO 9000 Quality Systems Handbook. Butterworth-Heinemann (Elsevier); 6 ed. Burlington, MA, USA

ISO 9000 (2005), http://www.iso.org/iso/qmp

Juran, J. M., (1999). Quality Control Handbook (6ed), McGraw-Hill, New York, NY, USA

Kaisler, S. H. (2005). Software Paradigms. Wiley-Interscience, Hoboken, NJ.

Khoshafian, S. N., and Copeland, G. P. (1986). "Object identity," *Proceedings of ACM Conference on Object Oriented Programming Systems Languages and Applications*, Portland, OR, November 1986, 406-416.

Lunt, B.M., Ekstrom, J.J., Gorka, S., Hislop, G., Kamali, R., Lawson, E., LeBlanc, R., Miller, J., & Reichgelt, H. (eds.) (2008). Information Technology 2008: Curriculum Guidelines for Undergraduate Degree Programs in Information Technology, Association of Computing Machinery (ACM), IEEE Computing Society (IEEE-CS)

Meyer, B. (1988). Object-oriented Software Construction. Prentice Hall, New York, NY.

Pham, H. (2000). Software Reliability. Springer, Berlin, Germany.

Pollack, S. (Ed.). (1982). Studies in Computer Science. Mathematical Association of America, Washington, DC.

Raymond, E. S. (1996). The New Hacker's Dictionary, 3rd ed. The MIT Press, Cambridge, MA.

Sales and Supply of Goods Act 1994, Ch 35, Legislation of Her Majesty's Government, The National Archives, UK, http://www.legislation.gov.uk/ukpga/1994/35/introduction

Scott, M. L. (2006). Programming Language Pragmatics, 2nd ed. Morgan Kaufmann, Maryland Heights, MO.

Shackelford, R., Cross, J.H., Davies, G., Impagliazzo, J., Kamali, R., LeBlanc, R., Lunt, B., McGettrick, A., Sloan, R., & Topi, H., (2005). Computing Curricula 2005: The Overview Report, Association for Computing Machiner (ACM), The Association of Information Systems (AIS), The Computer Society (IEEE-CS)

Shneiderman, B. (1992). Designing the User Interface: Strategies for Effective Human-Computer Interaction, 2nd ed. Addison-Wesley, Reading, MA.

Soldan, D., Hughes, J.L.A., Impagliazzo, J., McGettrick, A., Nelson, V.P., Srimani, K., & Theys, M.D. (eds.) (2004). Computer Engineering 2004: Curriculum Guidelines for Undergraduate Degree programs in Computer Engineering, IEEE Computer Society (IEEE-CS), Association for Computing Machinery (ACM)

Topi, H., Valacich, J.S., Wright, R.T., Kaiser, K.M., Nunamaker, J.F. Jr., Sipior, J.C., & de Vreede, G.J. (eds.) (2010). IS2010: Curriculum Guidelines for Undergraduate Degree Programs in Information Systems, Association for Computing Machinery (ACM), Association for Information Systems (AIS)

Van Vliet, H. (2008). Software Engineering: Principles and Practice, 3rd ed. Wiley, Hoboken, NJ.

Waguespack, L. J. (2008). "Hammers, Nails, Windows, Doors and Teaching Great Design," *Information Systems Education Journal*, 6 (45). http://isedj.org/6/45/. ISSN: 1545-679X

Waguespack, L. J. (2009). "A Two-Page "OO Green Card" for Students and Teachers," *Information Systems Education Journal*, 7 (61). http://isedj.org/7/61/. ISSN: 1545-679X

Waguespack, L. J., Schiano, W. T., Yates, D. J. (2010a). "Translating Architectural Design Quality from the Physical Domain to Information Systems," *Design Principles and Practices: An International Journal*, 4, 179-194

Waguespack, L. J. (2010b). Thriving Systems Theory and Metaphor-Driven Modeling, Springer, London, U.K.

Waguespack, L. (2011). "Design, The "Straw" Missing From the "Bricks" of IS Curricula," *Information Systems Education Journal,* 9(2) pp 101-108. http://isedj.org/2011-9/ ISSN: 1545-679X

Zuse, H. (1997). A Framework of Software Measurement. Walter de Gruyter, Berlin, Germany.

Appendix A – Choice Properties (Waguespack 2010b)

	Choice Property	Modeling Action	Practical Action Definition
1	Stepwise Refinement	elaborate	develop or present (a theory, policy or system) in detail
2	Cohesion	Factor	express as a product of factors
3	Encapsulation	encapsulate	enclose the essential features of something succinctly by a protective coating or membrane
4	Extensibility	extend	render something capable of expansion in scope, effect or meaning
5	Modularization	modularize	employing or involving a module or modules as the basis of design or construction
6	Correctness	align	put (things) into correct or appropriate relative positions
7	Transparency	expose	reveal the presence of (a quality or feeling)
8	Composition of Function	assemble	fit together the separate component parts of (a machine or other object)
9	Identity	identify	establish or indicate who or what (someone or something) is
10	Scale	focus	(of a person or their eyes) adapt to the prevailing level of light [abstraction] and become able to see clearly
11	User Friendliness	accommodate	fit in with the wishes or needs of
12	Patterns	pattern	give a regular or intelligible form to
13	Programmability	generalize	make or become more widely or generally applicable
14	Reliability	normalize	make something more normal, which typically means conforming to some regularity or rule
15	Elegance	coordinate	bring the different elements of (a complex activity or organization) into a relationship that is efficient or harmonious

Appendix B - OO Green Card (Waguespack 2009)

The OO Paradigm

Without a Language or Syntax!

What is the object world all about?

The Object-Oriented System Ontology

This ontology is consistent with the practice in computer science and information science categorizing a domain of concepts (i.e. individuals, attributes, relationships and classes). In this ontology of the object-oriented paradigm I attempt to minimize the vestiges of implementation languages and development methodologies in order to expose the core nature and value of object-oriented concepts.

1. Individuals

The most concrete concept in the object-oriented paradigm is the ***object***. It derives from the living physical experience of humans seeing and touching things. In that experience objects are separable – distinguishable from other objects by nature of their physical presence and location regardless of any other discernible characteristics they may possess. This characteristic of "individual-ness" leads to the property of ***identity***. Identity enables the unambiguous designation or selection of every object physical or abstract within a domain of discourse.

Objects have an "inside," an "outside," and a "surface" that separates the inside from the outside. An object contains anything that exists on the "inside" of the object. Since the surface of most physical objects is opaque, usually the contents are invisible and untouchable by anyone on the outside. This property renders the object's contents impervious to meddling and is called ***encapsulation*** (or ***information hiding*****).**

2. Attributes

Attributes are those characteristics that are inherent to an *object*. In the object paradigm attributes define either data or behavioral characteristics - each of which has a static and dynamic form. Attributes in static form combine to define what is called the *structure* of an *object*. From inception to extinction the *structure* of an *object* is immutable.

2.1. Data Attributes

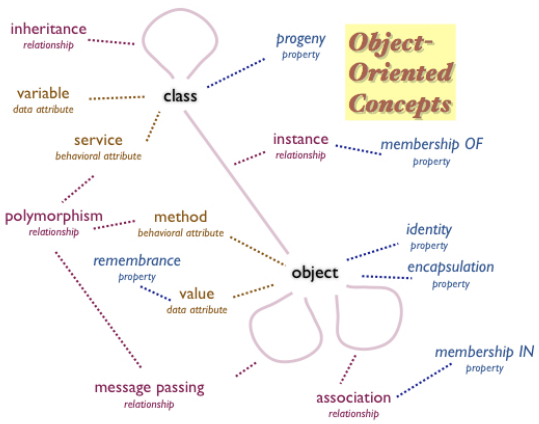
Data attributes serve to store information (data) within an *object* and implement the property of ***remembrance***. Data attributes are completely contained within an object protected by *encapsulation*. *Remembrance* is manifest statically as "what *can* be remembered," a ***data attribute variable***. It is manifest dynamically as a definition of "what *is* remembered," a particular ***data attribute value***.

2.2. Behavioral Attributes

Behavioral attributes serve to define the animate nature of an *object*. In its static form each behavioral attribute defines "*what* an object can do," usually called a ***service***. In its corresponding dynamic form this behavioral attribute defines "*how* a service is accomplished," usually called a ***method*** (or ***operation***). *Methods* define "activity" performed in an object model. A *method* may simply be access to *remembrance* inside an object or it may be complex sometimes employing the involvement of other *services* of the same or other objects to accomplish its responsibility. *Methods* reside within the *object* subject to *encapsulation* while *services* are visible at the surface of the *object* available for collaboration.

3. Classes

The ***class*** concept combines both a definition of *structure* and the generation of *object(s)* based on that *structure*. Every *object* is an ***instance*** of a specific *class* and shares the same static *structure* defined by that *class* with every other *object* of that *class*. The responsibility of generating *instances* that share the same *structure* is the property of ***progeny***. The *class* concept thereby fuses the existence of the *objects* to that of their *class*; *objects* cannot exist independent of their defining *class*. *Objects* are said to be ***members of*** their *class*.



Along with the static behavioral structure of *service* defined in the *class*, the dynamic behavioral attribute, *method*, may also be defined. Defined in the *class* this dynamic behavioral attribute, "*how* a service is accomplished," is identical for each and every *object* generated of that *class*.

4. Relationships

Relationships in the object paradigm exist on two dimensions: structural and behavioral. The structural relationships are based primarily on the properties of *identity, remembrance* and *progeny*.

4.1. Structural Relationships

4.1.1. Inheritance

Inheritance is a relationship between *classes*. The *structure* defined in one *class* is used as the foundation of *structure* in another. By foundation it is meant that all the *structure* of the first is replicated in the second and additional *structure* in terms of *data attributes* or *services* may be added or *methods* for replicated *services* may be altered (***overridden***). The replicated *structure* defines how the two *classes* are alike. The additions or alterations define how they are different. The *class* defining all the *structure* shared between them is called the ***parent class*** (*super class*, *generalization*) while the other is called the ***child class*** (*sub class*, *specialization*). It is said that the *child class* proceeds from or is derived from the *parent class*. Successive application of *inheritance* defining related *classes* results in a ***class hierarchy***.

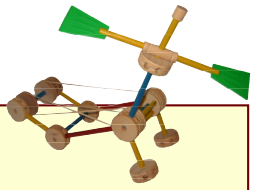
4.2. Behavioral Relationships

The behavioral relationships are based primarily on the property of ***membership IN***, and the capacity of *objects* to "act."

4.2.1. Association

An ***association*** is a relationship between *objects*. *Objects* are intrinsically separable by way of the *identity* property. At the same time, humans are compelled to categorize their experience of things in the physical world. Humans superimpose groupings that collect *objects* ***into*** sets (a foundation of mathematics based on human experience). *Objects* become members in a group only by designation. This property is called ***membership***. *Membership* is independent of *identity* or *attribute*. This property also permits humans to identify an *object* that is not in a set (i.e. discrimination). (*Membership in a group is discretionary and is distinct from membership of a class which is intrinsic by way of progeny.*)

Variations on *membership* derive from the intent of the relationship and generally fall into the categories of *association* and *composition*. Any designated collection of objects defines a relationship between those *objects* called ***association***. By the simple fact that they are members in the same relationship that membership defines how they relate. When the existence of the *objects* themselves is coupled with their membership; that is to say, if one (or the other or both) would not exist if it were not related to the other then the relationship is called a ***composition***.



Without syntax?

Every *language* that is invented to express concepts carries with it the understanding and the biases of the inventor. Depending on his/her purpose(s) those biases simplify certain tasks performed with the language but may obscure the underlying concepts.

Programming language design must deal with the feasibility of automated translation and interoperability with other programming languages and operating systems. Compromises and assumptions are chosen to make the resulting language efficient, effective and marketable.

The goal of this description of the object-oriented paradigm is to succinctly make the concepts understandable - an ambitious task to say the least!

- Professor Waguespack

4.2.2. Message Passing

Message passing is a relationship between *objects*. *Message passing* relies on the *identity* property and *services*. A ***message*** is a communication between a ***sender*** *object* and ***receiver*** *object* where the *sender* requests that the *receiver* render one of its *services*. The *sender* and *receiver* may be one in the same *object*. The *message* designates the *receiver*'s *identity*, the *receiver*'s *service* to be performed along with any parameters that the *service*'s protocol may require. Since the *message* is a request there are no implicit timing constraints determining when the *service* is accomplished. Unless explicitly designated a *message* results in an asynchronous activity on the part of the *receiver* without acknowledgment or returned information.

4.2.3. Polymorphism

Polymorphism results from the interplay of *message passing*, *behavioral attributes* and *classes*. A *sender* directs a *message* to a *receiver* designating a *service* of that *receiver*. A *message* does not designate a *method*. The regime that determines which *method* satisfies a service request is called ***binding***. If the *method* (corresponding to the *service*) is defined in the *class* of the *receiver object*, that *method* is invoked. If the *service* of the *receiver*'s *class* is *inherited* (and not *overridden*), the corresponding *method* defined in the nearest progenitor (*parent class*) of the receiving *object*'s *class* is invoked.