Is Object-Orientation Optimal for Modeling Information Systems?

Full Paper

Gary Spurrier  
Bentley University  
gspurrier@bentley.edu

Heikki Topi  
Bentley University  
htopi@bentley.edu

Abstract

In recent years, object-oriented (OO) systems development models, methods, and techniques have become pervasive. Via innovations including integration of data and behavior, encapsulation, polymorphism, and inheritance, OO continues to promise significant improvements over structured methods. However, research into OO's effectiveness has produced equivocal results. Further, research suggests that the standard OO grammar, the Unified Modeling Language (UML), is flawed, potentially explaining why UML use is relatively low and inconsistent. Given these findings, we propose a theory that OO itself may be sub-optimal for the analysis and design of broad classes of information systems (IS), including transaction-processing and decision support. This is based on the insight that many information systems entities inherently include either behavior or data, but not both, thus bringing into question the OO assumption that everything in a problem domain should be modeled as a domain object. From this, we propose implications for practice and research.

Keywords

Object-orientation, conceptual modeling, information systems analysis and design, ontologies

Introduction

Object-orientation (OO), including both OO analysis and design (OOA&D) and OO software development, has become pervasive in today’s information systems (IS) environment. Practitioners view OO techniques as a potential solution to the general challenge of delivering effective solutions in a timely and efficient manner (Sheetz et al. 1997).

As an example of OO’s ubiquity in analysis and design, the Unified Modeling Language (UML) has been adopted as an OO standard by the Object Management Group (OMG), and object-oriented SA&D is nearly universally taught in undergraduate and graduate IS programs (Dobing and Parsons 2009, p. 271). Based on our analysis of publicly available SA&D course syllabi, it has at many prominent universities (such as Carnegie Mellon University, Indiana University, University of Arizona, and UT Austin) replaced the older, structured analysis and design approach that utilized flowcharts, DFDs, and ERDs (Barros and Hofstede 1998).

Further, dominant software programming languages today, including Java, C#, C++, and Python, are inherently object-oriented, meaning that programs must be written in terms of object classes for them to even compile. Simple scripting languages, such as JavaScript, also have features that are object-oriented in nature.

The purpose of this paper is to explore the effectiveness of OO for analysis and design and, based on well-known ontological theories of information systems modeling, introduce an initial theory that develops further the idea that the type of an information system being developed determines whether OO should be used for analysis and design. We propose a range of related research questions that we believe the IS community should systematically explore.
Concerns over the Effectiveness of OO

Evidence for the effectiveness of OO and its associated modeling standard, UML, is, however, sketchy, particularly outside the realm of programming. In this section we will discuss research findings supporting this claim.

Gelbard et al. (2010) note that UML diagrams other than class diagrams and interaction diagrams are perceived as not user friendly. This is true for developers, and even more so for analysts and users. Gelbard et al. further suggest that class diagrams should include only data items, not methods. In effect, this reduces them to entity relationship diagrams (ERDs) (Parsons, 2002). Gelbard et al. (2010) point to issues pertaining to the clarity, integrity, and completeness of UML diagrams, especially for analysis, which may help explain why they are not used more frequently and consistently.

In general, UML appears to suffer from multiple problems arising from its creation in the late 1990s as a conglomeration of practices from competing OO modeling techniques. The resulting grammar is at once excessively complex, lacking in flexibility, and lacking consistent guidance in its adoption and use (Dobing and Parsons 2008). In addition, it is based on principles that are ontologically unsound, as will be further developed below. Adoption of UML by practitioners has, perhaps unsurprisingly, been found to be spotty (Davies et al. 2006; Petre, 2013).

Analysts and developers also often find learning OO to be challenging (Sheetz et al., 1997), including both novices as well as experts who have expertise in structured approaches (Nelson et al. 2009). Indeed, this difficulty appears to be the case regardless of any potential interference from prior knowledge of structured development techniques (Shaft et al. 2008).

In addition to the challenges related to usability of OO modeling grammars, research evidence points to other problems with OO approaches. Evidence supporting the claim that OO promotes software reuse is weak (Irwin, 2002), in the case when reused models and components require modification. This is true both in the context of analysis and design and in the reuse of actual code. Despite the pervasive use of OO for programming, persisting data in information systems continues to be dominated by relational database management systems (RDBMS) (DB-Engines 2017). Given that OO programs structure data somewhat differently than the relational model used for data storage, there is an ongoing need for IT practitioners to solve this issue using object-relational mapping techniques (Larman 2005, p. 622).

Further, while we have noted that OO, in general, has become pervasive, particularly in implementation, it is also the case that a number of data-oriented tools remain non-OO. For example, SQL and newer tools such as the “big data” Hadoop projects Pig and Hive are not OO, when viewed from an analyst user perspective (White 2015, pp. 423, 471). None of the most widely used NoSQL databases (DB-Engines 2017), document store MongoDB, wide-column store Cassandra, and key-value store Redis, are specifically object-oriented in nature (see, e.g., MongoDB 2010).

Explaining OO as a Pervasive Phenomenon

Given the issues noted above, one might wonder how and why OO has become so pervasive. Although not the main focus of this paper, we suggest that OO techniques were found to be effective in the context of rich graphical presentation layers and, from there, their use spread to the business logic layer. The use of OO in this expanded context may, however, often not be necessary nor effective at all.

The rise of rich, desktop and browser-based GUIs in the 1990s provided an impetus for software development environments and approaches providing support at the presentation layer for many classes (literally) of user interface objects that included both attributes and methods (e.g., text boxes, radio buttons, check boxes and check box lists, dialogue boxes, and so on). OO provides mechanisms that have a strong natural fit with these environments.

However, the value of extending the use of OO as a standard for business logic and persistent data layers is less certain, as per the research noted above. It may be that OO has been widely adopted partly from marketplace promotion and fashion and partly because it came to be seen as an inevitability. For example, focusing on the phenomenon of agile development techniques, Cram and Newell (2016) argue that “new organizational techniques are influenced by factors including the rhetoric of influential gurus, research funding bodies, and practitioner demand for innovative solutions to practical problems,” such that new
approaches such as agile may literally be seen as a management fashion. Thus, it may be that, in part, OO has been similarly adopted as a fashion, particularly because it offers the promise of seamless integration between analysis, design, and implementation. Further, it may be that as OO began to be perceived as becoming dominant, practitioners stopped asking “should we adopt it” and began asking “how do we adopt it” (Cho and Kim, 2002).

**OO And Structured Techniques As a Continuum**

Sircar et al. (2001) note both differences and similarities between OO and the older, structured techniques that OO has displaced. In terms of differences, OO models domains and develops software utilizing classes of “objects” that include both data and logic, while structured models consider data and the logic operating on that data as separate things (Hoffer et al. 2011, pp. 302, 532).

However, in numerous other areas, OO and structured techniques share patterns, including, for example, the core control structures within functions or methods such as the use of sequence, iteration, and conditional decision. Further, some other concepts often considered unique to OO can be found in structured approaches. For example, encapsulation is a specific form of the more general idea of information hiding within loosely coupled functions, while polymorphism is a more specific form of the general idea of function overloading. Inheritance of attributes is present as a concept in many non-object-oriented grammars, such as the Enhanced Entity-Relationship model.

Thus, in large measure, the fundamental difference between OO and structured techniques can be considered to be in OO’s insistence that both the solution domain and the problem domain be modeled as classes of objects that integrate data and logic. The question, then, becomes “Is this approach of viewing the world exclusively in terms of ‘objects’ optimal in all circumstances?”

**Ontological Theories of Information Systems Modeling**

In evaluating IS techniques (and especially modeling techniques), a significant amount of research is built on the foundation of ontology, which is an area of study concerned with fundamental constructs of reality (Recker et al. 2009). A key question is whether these ontological frameworks actually tap into all the key issues driving the appropriateness of OO for MIS.

**Key Ontological Frameworks Used in Research**

Frameworks based on the foundation of ontology have been used by many researchers to evaluate IS design approaches, including user requirements, design artifacts, and development practices (Irwin and Turk 2005).

A key example is the Bunge-Wand-Weber (BWW) representation model. BWW holds that modeling techniques in IS can be evaluated in terms of their ability to portray fundamental ontological concepts: Things, States of Things, Events and Transformations occurring to Things, and Systems structured around Things (Recker et al. 2009). According to this theory, modeling techniques should exhibit completeness in modeling the key ontological concepts, as well as clarity, which is defined as avoiding construct overload, construct redundancy, and construct excess between reality and the model (Wand and Weber 2002). However, given that we have established above that both OO and structured approaches share these same core concepts, albeit organized differently, it is not clear if the BWW representation model can provide much guidance in our current research context.

Rather, we suggest that another framework, the Good Decomposition Model (GDM), may be more pertinent, although—as explicated below—not completely to the point. GDM holds that decomposed models can be evaluated in terms of five desirable factors: Minimality (no redundant or unused variables), Determinism (system is predictable with respect to its response to events), Losslessness (emergent properties are explicitly represented), Minimal Coupling (modules should have minimal external interactions), and Strong Cohesion (modules should have high internal integration) (Weber 1997). We will return to GDM—and, in particular, its concepts of Weak Coupling and Strong Cohesion—as we explicate our theory, below.
In Reality, Is Every “Thing” an “Object”?

Although perhaps not completely salient in these ontological frameworks, we can discern the issue of whether methods and data should always be combined into objects. For example, Irwin & Turk (2005, p. 19) note that even in the BWW representation model, “processes, or transformation laws, are not things in and of themselves and thus do not themselves have properties or attributes...Instead, a process or transformation law is an algorithm that changes the state of one or more things. This ontological perspective differs from an object-oriented perspective, where we often create process-type objects with behavior and attributes.”

This viewpoint contrasts with Agarwal et al. (1996), who argue that process-oriented methods place greater emphasis on processes while OO methods place more emphasis on data and structural relationships. In fact, for applications that involve large amounts of persistent data, process-oriented methods (such as DFDs) are generally used in conjunction with data-oriented methods (such as ERDs) (Hoffer et al. 2011, p. 22), such that OO and traditional methods place an equal level of emphasis on methods and data, just with different approaches to integrating the two.

Objects vs. Data Entities vs. Logic Entities

To make sense of where OO fits well vs. where it doesn’t, we therefore need to expand the ontological notion of a problem domain “Thing.” We propose three general categories: Objects, Data Entities, and Logic Entities.

“Objects” are Things with both domain logic and persistent data. They are often, but not always, tangible objects (or interacting aggregations of tangible objects) executing sensorimotor behaviors in real space and time, for example, robots, unmanned vehicles (UVs), or even software-enabled home appliances such as refrigerators. However, they can also be intangible, as, for example, daemons.

“Data Entities” are Things with persistent data but no inherent domain logic. There are many examples in the realm of information systems, including objects representing transactions (e.g., orders, payments, healthcare claims) or aggregated management information (financial statements, cost trend data, and so on). We also include as Data Entities “logic tables,” which are series of parameter values controlling aspects of the execution of logic.

With respect to Data Entities, it is worthwhile noting that, in their study analyzing Business Process Modeling Notation (BPMN) constructs against the BWW framework, Recker et al. (2010) confronted the BPMN construct “Data Object,” which represents a document or data record. The researchers argued as to whether a Data Object represented a “construct excess” in the BWW framework, but ended up concluding that it mapped to the concept of a “Class of things” (albeit, ones without inherent behavior).

Finally, “Logic Entities” are domain-specific procedures, functions, routines, components (groups of routines), or subroutines that take input, manipulate data, and generate output, but do not themselves include persistent data. Logic Entities therefore provide a means of applying a business algorithm to data, specifying a set of rules for transforming the state of an instance of a Data Entity or a collection of Data Entity instances. There are many practical contexts in which this type of pure computational capability is needed. Consider, for example, in the pre-electronic computer era, where a human being could literally be labeled as a “computer” (Grier, 2005, pp. 1-7), engaged in routine, computational problem-solving. The point is that the data inputs and outputs recorded did not include the “human computer,” much as an electronic software method is generally not recorded.

Example of Common Difficulties in OO Modeling for IS

It is easy to create an example illustrating how difficult it is to model a highly common IS problem domain using OOA&D techniques. Consider the problem of summarizing raw data to information, which is perhaps the most generic definition of a management information system problem domain possible. Specifically for the sake of illustration, we take the highly simplified case of summarizing raw healthcare claims data to cost trend information by month and by patient coverage group. As shown in the simple class diagram in Figure 1, the problem domain consists of two domain entities: the raw data, “Healthcare Claim,” and the summarized information, “Cost Trend,” which we want to maintain in order to make any
possible adjustments permanent, perform future period modeling, and so on. The logic to perform the summarization is represented by the method “summarizeByMonthByGroup().” The simple question then is: into which domain entity should this method be allocated? Or do we need to create an artificial class to house the method? Object modeling argues for including each method within the representation of the domain entity it is related to. But that decision is difficult in this case, as simple as it is. For example, it is illogical to place the method in the Cost Trend class, as these instances are literally created by the method, itself. But the alternative of placing the method in the Healthcare Claim class is problematic, as well, as the method needs to iterate over the overall group of Healthcare Claim objects (in as sense, operating outside any given claim) in order to create the summary Cost Trend objects.

![Figure 1. UML Class Diagram Illustrating Difficulty in Using OO to Conceptually Model the Generic Problem of Summarizing Raw Data to Management Information](image)

Similar modeling challenges can easily be constructed in the context of transaction processing, although we omit such an example here due to space constraints.

As we shall see in the next section, the OO community has addressed issues such as this one with “design patterns.” But, in pragmatically solving such challenges, those design patterns deviate significantly from “pure” OOA&D principles. It is our belief that these challenges arise directly from mismatches between general characteristics of IS domains and the core assumptions of the OO paradigm—specifically, the existence of many IS problem domains where the domain is better modeled using separate Data Entities and Logic Entities than with Objects.

Indeed, consistent with this line of reasoning, there is little or no evidence supporting the proposition that integrating Data Entities and Logic Entities will lead to a superior outcome in contexts where the two do not naturally belong together.

**Insights From OO Design Patterns**

Over time, the OO community has developed a series of software design patterns, where “software design pattern” is understood to be an effective, general approach to addressing common challenges arising in software design and development (Larman 2005, p. 4). Interestingly, a careful reading of these sources reveals acknowledgements that not all business domain entities can best be viewed as Objects, as defined above. Rather, we can identify general, common information systems situations in which Data Entities and Logic Entities should be modeled separately, even in an otherwise OO approach.

**OO Design Pattern Examples for Data Entities**

Given the need to manage large amounts of persistent data from RDBMS, a common practice advocated in numerous OO programming texts is for the OO application to issue an embedded SQL-like query to the RDBMS, returning a table of structured data commonly called a “result set” (in a Java context, for
example, see Bai 2011, p. 140; Murach and Urban 2014, p. 368; and Savitch 2016, p. 1120). While a result set is considered an object in OO languages, it includes no business domain-specific logic (rather, it only includes generic, built-in methods to do things like move to the next row, retrieve a value, and so on). As such, in our framework, we would consider a result set to be a Data Entity, not an Object.

A key question then arises: should the result set then be “materialized” (to use the OO term) into a collection of individual objects (e.g., a linked list or hash set), such that each record could have its own OO methods, as per the core idea of OO? Interestingly, several OO design pattern sources advocate, at least in some cases, for just working directly with collections of data in the form of a single object containing the result set, without going to the trouble of translating that result set into a collection of individual objects.

For example, Marinescu (2002, pp. 64-65) argues that translating result set rows to a collection of objects is redundant and, in fact, less straightforward if the task requires pushing tabular data to a client. Similarly, Fowler (2003, pp. 27-28, 124-125) presents the approach of what he terms a "table module," a single object instance that handles the business logic for all rows in a result set, as being as worthy of consideration as a "domain model," in which each record is instantiated as a separate object (and likely organized into a collection). Finally, Ambler (2013) notes that, although it “might sound like blasphemy to object purists, you don’t always need nor want objects as the result of a search.” Acceptable alternatives include result sets and "data transfer objects," collections of objects that just contain data with basic getters and setters to access the data, but no business logic.

**OO Design Pattern Examples for Logic Entities**

Similarly, when logic is not tightly coupled to persistent data, OO design patterns endorse creating what are, in effect, Logic Entities.

For example, Fowler (2003, pp. 132-138) suggests using a set of "service layer" objects to house "application logic," which is defined as business logic that does not fit well into domain objects. This includes things like communications operations and calculations that would tend to be duplicated in regular domain objects and would, hence, be less easily reusable. Similarly, Ambler (2014) advocates for several interacting "layers" of classes. For example, in addition to the regular domain classes, he describes a process layer of classes that “implements business logic that involves collaborating with several domain classes or even other process classes.” Finally, Larman (2005, pp. 424-425) distinguishes between classes derived from “representation decomposition” vs. those derived from “behavioral decomposition.” The former represent traditional OO “things in a domain,” while the latter are “algorithm objects” not corresponding to a “real-world domain concept.” Rather, these algorithm objects are "Pure Fabrications" of the software designer, using Larman’s term for this design pattern.

**Example Revisited in Terms of Implemented IT Artifacts**

Via these OO design patterns, we can discern how OO developers circumvent the limitations of OO for conceptual modeling of information systems. Plainly put, they violate OO principles, in essence turning to structured approaches using OO wrappers to enable their programs to compile. For example, returning to the Healthcare Claim summarization problem shown in Figure 1, a developer could create a solution in several ways, but the most straightforward approach in light of these design patterns would be to create a ClaimSummarization class that includes a static “summarizeByMonthByGroup()” method (implemented, for example, with the Stream API in Java) and the inclusion of two result sets: HealthcareClaim, containing the raw claim data queried from the RDBMS, and CostTrend, which would receive the summarized output generated to then be persisted to the RDBMS.

The key point here is that this is only nominally OO. Put another way, the method iterating over the raw healthcare claims is really equivalent to a structured function or sub-routine—a Logic Entity in our extended ontology. Further, the selected healthcare claim records and output cost trend records are pure data without inherent domain behaviors—Data Entities in our extended ontology.

**The Issue of Tangibility vs. Intangibility**

Research (Soffer and Hadar 2007) indicates that modelers find it easier to model tangible domains (e.g., trucks, shipped items) than intangible domains (e.g., university courses, course sections). It often seems
Is Object-Orientation Optimal for Modeling Information Systems?

to be the case that IS, where our concerns over OO mainly lie, are intangible, while systems of “Objects” are often tangible. However, this is not consistently the case, and (in)tangibility is not the key to our theory. Rather, we focus on whether a “Thing” has logic and/or data (and ontologically, we view that if a “Thing” possesses neither, then for the purposes of IS “It” does not exist). See examples in Table 1.

<table>
<thead>
<tr>
<th>Intangible</th>
<th>Tangible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retail order</td>
<td>Hardware item</td>
</tr>
<tr>
<td>Healthcare claim</td>
<td>Food</td>
</tr>
<tr>
<td>Logic table</td>
<td>Fuel</td>
</tr>
<tr>
<td>Healthcare claim benefits determination</td>
<td>Computation via “Human Computer”</td>
</tr>
<tr>
<td>Summarizing raw data to information</td>
<td>Medical examination</td>
</tr>
<tr>
<td>ETL map to transform data formats</td>
<td>Machinery adjustment</td>
</tr>
<tr>
<td>Software daemon</td>
<td>Robot</td>
</tr>
<tr>
<td>Web service</td>
<td>Unmanned Autonomous Vehicle (UAV)</td>
</tr>
<tr>
<td>Newsfeed (e.g., RSS aggregator)</td>
<td>“Thing” in Internet of Things (e.g., intelligent household appliance)</td>
</tr>
</tbody>
</table>

Table 1. Examples of Intangible and Data Entities, Methods, and Objects

Theory: Where OO Fits and Where It Does Not

Revisiting Good Decomposition Model (GDM) Concepts

The examples above represent common, general IS situations in which logic may or may not be tightly coupled to individual instances of data. This, in turn, suggests separating data from logic (e.g., concretely, as result sets and service layer objects), even in an otherwise OO context.

To a degree, this is suggestive of the GDM ontological approach introduced above. One could argue that, in these cases, logic and data are in a sense Weakly Coupled, such that segregating them promotes High Cohesion and, it follows, promotes reusability. This is, in fact, a benefit that is noted in discussions of OO patterns (Larman, 2005, p. 425). One could even argue that this segregation promotes a form of GDM Minimality, although in this case, the minimality is in the form eliminating redundant instances of logic, rather than data (“state variables”).

Theory Implications of Fundamental Ontological Assumptions

However, our view—and, thus, our theory—is that the issue is more fundamental than one of modeling. It is really an issue of the universal appropriateness of the core ontological assumption of the OO paradigm. Specifically, the OO paradigm assumes that things in the real world have both behaviors (transformation logic) and properties (data), such that it is most natural and, thus, effective, to treat them as Objects.

We, in contrast, in contemplating typical types of information systems, observe that this assumption is often not true. That is, many IS are dominated by things that are either Data Entities or Logic Entities, but not Objects. Note that the latter is typical of two of the most important IS system categories (Clark 1992):
• **Transaction processing:** Where a single Data Entity instance (a database record) is being added or updated, often via complex Logic Entity(s) depending on data from other Data Entities.

• **Decision support:** Transformation of raw data to summarized information (intrinsically involving multiple types of Data Entities).

We therefore hypothesize that OO fits best where the problem domain is dominated by “Objects,” as defined above. In contrast, OO will be less natural and effective where a problem domain is dominated by “Data Entities” being operated on by a series of “Logic Entities.”

It follows that, for many, if not most, IS, OOA&D may be of little benefit and may, in fact, be detrimental to the modeling effort, and perhaps ultimately to the development of the executable software product. Thus, we argue that the OO approach should extend, not replace, the structured approach.

**Practice and Research Implications**

These hypotheses obviously have major implications for both researchers and practitioners. If they are correct, then many practitioners have been utilizing a sub-optimal approach for modeling and constructing information systems.

Exploring this from a research perspective suggests a major program of investigations into the areas of problem domains and how they should be modeled.

**Research Gaps in Modeling Problem Domains**

There has been a great deal of prior research on conceptual modeling, but none known to us that manipulate the ontological issue of Objects vs. Data Entities vs. Logic Entities. For example, Topi and Ramesh (2002) published a comprehensive review of literature on usability of conceptual data modeling approaches and techniques, and this issue was not addressed in their review at all. Other studies have examined related issues, but none that squarely address these problem domain ontological issues. For example, Khatri et al. (2006) examined analyst level of domain knowledge. Marakas and Elam (1998) examined analyst questioning techniques. Liang et al. (1998) focused on identifying objects based finding “purposeful activities” (but seemingly to the exclusion of examining data and, hence, “Data Entities”).

**Calls for Research Modeling Problem Domains**

However, some researchers have suggested that the problem domain is an important aspect to study for modeling. For example, Hickey and Davis (2004) argued for studying modeling business management information systems vs. engineering problems. Also, Siau and Rossi (2011) argue that selecting a modeling method should be “based on project contingencies such as...the problem under investigation,” given that “the organization of constructs defines the world view...and specifies the limits of what can be modeled with a given method.” This would seem to align with our proposition regarding OO being limiting in terms of modeling domains dominated by Logic Entities and Data Entities. And, finally, Burton-Jones et al. (2009) do hint at this issue with their concept of “pragmatics,” which includes “aspects of the task for which grammar or script is being used” (p. 22), but also includes other factors such as the level of knowledge of the modeler, herself or himself.

We certainly advocate for pursuing this type of research and recommend that the IS research community systematically address a broad range of related research questions, including but not limited to:

• Can we validate the ontological view of Objects vs. Data Entities vs. Logic Entities with practitioners?

• Given the discussion of OO design patterns, above, is a model implementing all of these concepts really OO? Or is it, rather, a superset of OO and structured techniques?

• Is the use of this “superset” consistent with practices of practitioners, even if they claim to be operating within the OO paradigm?

• Do practitioners utilizing the “superset” tailored to characteristics of the problem domain generate more successful systems than those operating in either a “pure OO” or “pure structured” paradigm?
• From the prior point, can we provide practitioners with any guidance as to what would be a "correct" (or, at least, "optimal") way to use OO and structured approaches to model and construct solutions in different problem domains?

• Is the “superset” something that we consistently teach (or should teach) in our professional programs, such that formally educated IT professionals could be expected to understand this as part of their standard set of skills?

Given the high importance of these issues, we hope that the rest of the field will join us in exploring these questions.

REFERENCES


