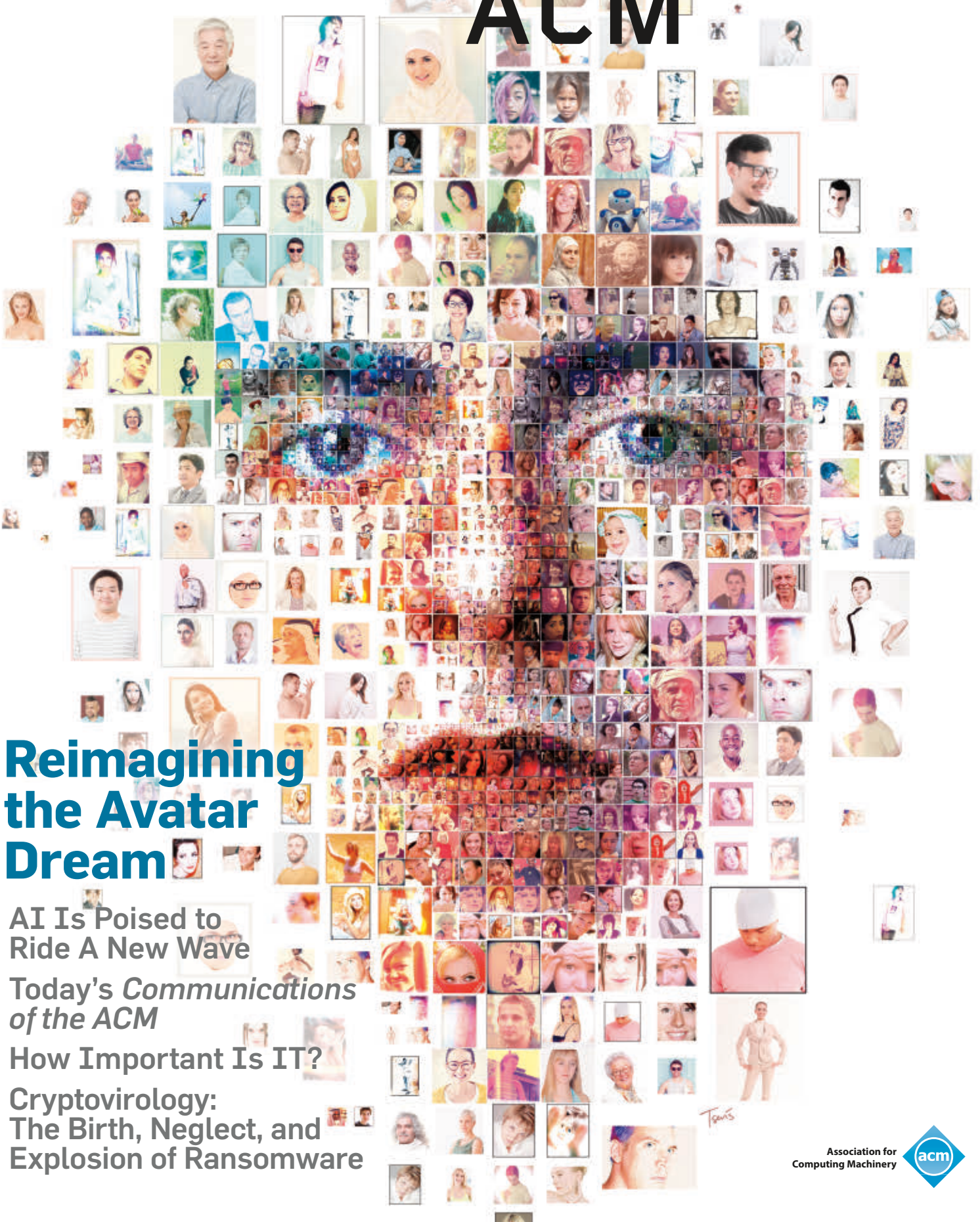
COMMUNICATIONS OF THE ACM

CACM.ACM.ORG

07/2017 VOL.60 NO.07



Reimagining the Avatar Dream

AI Is Poised to Ride A New Wave

Today's *Communications of the ACM*

How Important Is IT?

Cryptovirology: The Birth, Neglect, and Explosion of Ransomware

Association for Computing Machinery

DOI:10.1145/3079970

Article development led by acmqueue
queue.acm.org

An improvement over UML.

BY MARK A. OVERTON

The IDAR Graph

UNIFIED MODELING LANGUAGE (UML)[6] is the de facto standard for representing object-oriented designs. It does a fine job of recording designs, but it has a severe problem: its diagrams don't convey what humans need to know, making the diagrams difficult to understand. This is why most software developers use UML only when forced to.[1]

For example, the UML diagrams in Figures 1 and 2 portray the embedded software in a fax machine. While these diagrams are attractive, they do not even tell you which objects control which others. Which object is the topmost controller over this fax machine? You don't know. Which object(s) control the Modem object? You don't know.

People understand an organization, such as a corporation, in terms of a control hierarchy. When faced with an organization of people or objects, the first question usually is: "What's controlling all this?" Surprisingly, UML has no concept of one object controlling another. Consequently, in every type of UML diagram, no object appears to have greater or lesser control than its neighbors. This absence of a control hierarchy in software design does much harm in the following ways:

- Designs are difficult to understand. Showing no hierarchy is like portraying a corporation by drawing a line between every pair of employees who interact with each other. Such a chart would rapidly become incomprehensible spaghetti. An organizational chart is drawn as a control hierarchy for good reason: people can readily understand them, regardless of the corporation's size.
- Because any object can interact with any other object in any way desired, code structure slides into disorganization as people add features and interactions to objects during design and implementation.
- Maintenance becomes slower and more error-prone because learning curves are steeper. In addition, maintainers can and do insert hacks anywhere, causing code to decay.

These problems mean designs tend to become messy during both initial implementation and maintenance, resulting in more bugs and delays.

The Basics of an IDAR Graph

To be useful, a graph that portrays software design must communicate in a way that humans understand. An organization of objects in software is analogous to a human organization, and almost without exception, an organization of people is portrayed as a control hierarchy, with the topmost person having the broadest span of control. Based on this idea, Figure 3 is a simple IDAR graph that portrays part of the same fax machine design shown in Figures 1 and 2, but expressed as a control hierarchy.

PHOTO BY FABRIKA SIMF

In an IDAR graph, boxes represent objects. If a class has only one instance (the most common case), then the box is labeled with the class name. An arrow connecting two boxes means that the upper object commands (and thus controls) the lower object. Such command arrows always point down. In Figure 3, the `Fax` object is the topmost controller, which commands the `Receive` and `Send` objects, which in turn control `ImageProc` (image processing). `Command` arrows may be labeled with the names of commands that are sent. For example, `Fax` commands `Send` to `sendFax`. Note that `ImageProc` has two bosses. Having multiple bosses is uncommon in human organizations but is common (and encouraged) in software to prevent redundant implementations.

Objects need to communicate in more ways than commands. For example, they often need to exchange data and inform each other about events and results. In an IDAR graph, such non-command communications are called *notices* and are shown as floating arrows. For example, in Figure 3, `Send` tells `Fax` that transmission is done via the `done` notice.

Both commands and notices are merely method calls. This means that the public methods in each object are divided into two groups: commands and notices. Software designers must give careful thought to which methods will be commands, because they determine the hierarchy. Commands and notices have constraints, which are precisely defined later in this article.

A note about terminology: when you call a command (method) in an object, you are said to be *commanding* (or *sending a command to*) that object; when you call a notice in an object, you are *notifying* (or *sending a notice to*) that object.

More Features of IDAR Graphs

A graph of a design should portray other salient features, such as threading, data flows, and the use of indirection. The complete IDAR graph of the fax machine in Figure 4 exemplifies some of these additional features.

The horizontal line above the `Connect` and `Negotiate` boxes is analogous to a horizontal line in an organizational chart: it groups subordinates under their manager. In an IDAR graph, such a *rail* (as it's known) is more general, as it indicates that all objects above the line (called *superiors* or *bosses*) command all objects below it (called *subordinates* or *workers*). In this fax machine, two superiors (`Receive` and `Send`) command three subordinates (`Connect`, `Negotiate`, and `ImageProc`).

An object containing a thread is said to be *active* and is denoted with double vertical lines on each side of its box. This notation was taken from UML. In the fax machine design in Figure 4, `Fax` and `ImageProc` are active.

Indirect method-calls are indicated by a bubble (circle) placed on the tail of the appropriate arrow. Such indirection can be explicit in the source code or implicit using polymorphic inheritance. Indirection is commonly used for notices sent from a subordinate to one of multiple superiors, such as the `connected` notice in Figure 4 that is sent from `Connect` to `Receive` or `Send`.

A subsystem is a separate hierarchy (a separate graph) with a *subman* (subsystem manager) as its topmost object. A subman controls its subsystem and is portrayed as an elongated hexagon. In a graph that commands a subsystem, only the subman is drawn. For example, in Figure 4, `Printer` and `Scanner` are the submans of their re-

spective subsystems, which should be shown in separate graphs.

A dashed arrow denotes a data flow. Notice-arrows often parallel a data flow, because data flows are usually implemented using notices. An example is the `pixelRow` notice sent from the `Scanner` subsystem to `ImageProc`.

Notice that the names of some commands and notices in Figure 4 are prefixed with numbers. These optional sequence numbers show the order of actions composing an operation. In this case, they show the sequence of calls to send a fax. A copy of this graph could be enhanced to show the sequence for receiving a fax. Such annotated graphs replace sequence diagrams in UML. They are easier to understand because you can see which actions are commands versus responses, in addition to their order.

It might surprise you that the IDAR graph in Figure 4 is the same design as the UML diagrams in Figures 1 and 2. Compare these diagrams. In the IDAR graph, you can easily see which objects control which others, thus revealing how this design operates.

Four Rules

The principles underlying IDAR graphs can be expressed in the form of four rules. They form the acronym IDAR, the namesake of these graphs. The rules are:

▸ **Identify.** Each public method in an object is identified as either a command or a notice. From its caller's viewpoint, a notice only imports or exports needed information. A command may do anything.

▸ **Down.** When graphing the calls to commands among objects, the arrows must point down.

Figure 1. UML class diagram for a fax machine.

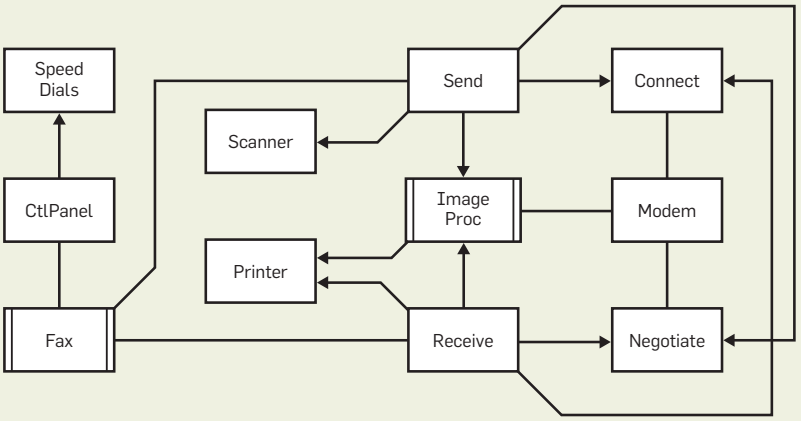
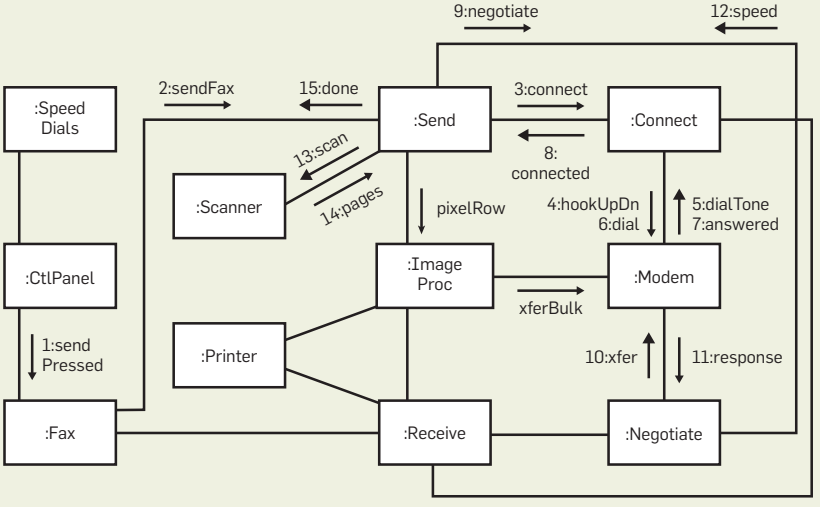


Figure 2. UML communication diagram for sending a fax.



▸ **Aid.** A command or notice may, unknown to its callers, aid its object by performing part or all of a previously commanded action.

▸ **Role.** A brief role is written for each object and method that summarizes the service it offers, avoiding any aspect of its implementation (including aid). Callers may rely on only what is stated in roles.

The Down rule ensures every design is a command hierarchy consisting of superiors and subordinates (bosses and workers). This rule produces a DAG (directed acyclic graph), so it's also known as the DAG rule. The Role rule requires that every method or object fulfill its role, doing no more and no less, precluding unexpected side effects. The Role and Down rules together force every design to be a role hierarchy. The Aid rule gives designers more flexibility by allowing public methods to help secretly with previously commanded duties, in addition to fulfilling their own roles. These rules don't apply to cross-cutting concerns.[2]

It's also helpful to think in terms of constraints on public behavior. Commands have one constraint: they must go down in the hierarchy (Down rule). Notices also have one constraint: they may only convey information (Identify rule).

Roles are important and warrant further discussion. A role is a purpose, responsibility, or duty. The Role rule requires that every object and method have a role that can be summarized in a few words, preferably containing only one verb. An example is: "Sends a fax." In an IDAR graph, the broadest role (greatest responsibility) is at the top, and the narrowest (most specialized) roles are at the bottom.

Inheritance creates a hierarchy, so why not use it? Unfortunately, inheritance creates a hierarchy of *categories*, which is less useful than a hierarchy of *roles*.

To see why, examine Figure 5, which shows a UML inheritance hierarchy for a CD player. `DiskMotor` and `LaserMotor` are subclasses of `Motor`, so they are in the motor category. You care little about their category, however, because you need to know which objects control these motors.

Likewise, `Laser`, `Motor`, and `Audio` are subclasses of `ElectronicDevice`, but that does not help because

you need to know which objects with broader roles command these devices. An inheritance hierarchy portrays categories, which are seldom helpful except in GUIs; it does not portray what you need to know—the hierarchy of roles.

Comparing UML to IDAR. An easy way to compare UML and IDAR is to follow an operation—for example, sending a fax. The sequence numbers on the commands and notices in Figure 4 indicate that after the user presses the `Send` button on the control panel, the `CtlPanel` object calls the `sendPressed` notice in `Fax`, which is clearly the main controller over the entire fax machine, and it commands `Send` to `sendFax`. Based on its high position in the hierarchy, you can see that `Send` handles the high-level aspects of sending faxes. It commands `Connect` to `connect` to the receiving machine, and `Connect` in turn commands `Modem` to take the phone off the hook via the `hookUpDn` method. After `Connect` gets the `dialTone` notice from `Modem`, it commands `Modem` to dial and waits for its answered notice. `Connect` then sends a connected notice back to `Send`. The figure also shows that `Send` commands `Scanner` to scan, and that data (the dashed arrows) will flow from `Scanner` into `ImageProc` and then into the `Modem` via the `pixelRow` and `xferBulk` notices. This graph reveals the structure of this software and how it works.

Figures 1, 2, and 6 are the UML class, communication, and sequence diagrams, respectively, for the same fax machine design. The communication diagram (Figure 2) has the same sequence numbers as the IDAR graph, making comparison easier.

Let's use the UML diagrams to show how a fax is sent. Which objects have primary roles? It's hard to tell. Which interactions among objects are the most important? It's hard to tell. Which objects are controllers versus workers? It's hard to tell. The best you can do is follow messages sequentially on the communication or sequence diagram, and even then it is difficult to determine which objects control which others, or which objects have broad versus narrow roles. UML fails to convey roles or their ranks, making designs hard to understand.

Benefits of IDAR Graphs

IDAR graphs provide several advantages over UML, two of which are predominant.

Ease of understanding. An IDAR graph is easier for developers to understand than the corresponding class, communication, and sequence diagrams in UML for the following reasons:

- The role hierarchy in IDAR is a generalized form of the AH (means-end abstraction hierarchy) employed in cognitive engineering,[7] which is known to impart understanding by means of the why-what-how triad. This triad consists of an object, its superiors, and its subordinates. It provides the following insights: the purposes of the object's superiors tell you *why* the object exists; the role of the object tells you *what* it does; and the purposes of its subordinates indicate *how* it works. UML lacks an AH, so it cannot tell you why an object exists or how it works.
- The hierarchy in an IDAR graph reveals which objects control which

Figure 3. Incomplete IDAR graph of a fax machine design.

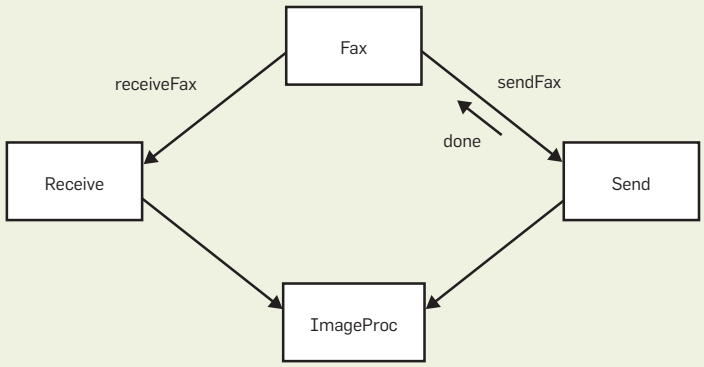
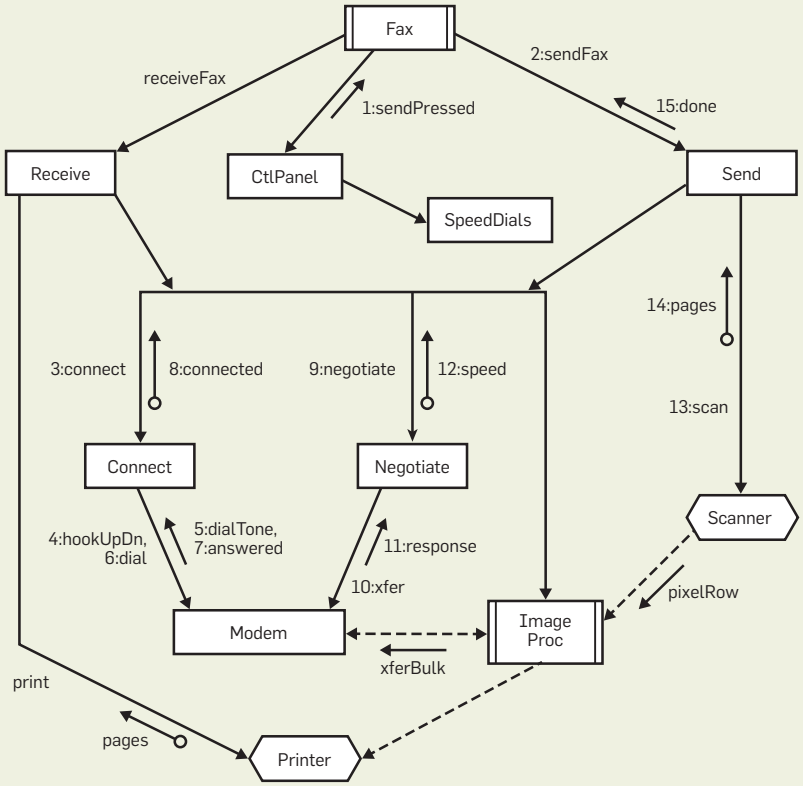


Figure 4. Complete IDAR graph of a fax machine.



others and, equivalently, which objects have broad versus narrow roles. In Figure 4, it is obvious that `Fax` is the top-level controller, and that `Send` and `Receive` are second-to-top-level controllers having rather broad roles. The corresponding UML diagrams conceal these helpful control relationships and role breadths.

▸ The subordinates of each superior form a closely related group, helping developers to associate functions with groups of objects. In Figure 4, it is clear that `Connect` and `Negotiate` are closely related workers under the same bosses, whereas UML conceals this tight affiliation. Unlike UML, IDAR reveals work groups.

▸ In an IDAR graph, command calls are more prominent than notice calls because they are more important. UML conceals degrees of importance.

▸ Throughout history, people have selected role hierarchies to represent organizations, indicating that they are most understandable.

▸ Research by experts in cognitive theory has shown that UML has severe problems with understandability ("cognitive effectiveness").[3] Specifically, UML has "alarmingly high levels of symbol redundancy and overload" and poor "visual discriminability." IDAR graphs were designed to avoid both of these problems.

▸ Other research has revealed that developers understand software design as an integrated interplay of its structure and behavior.[1] UML splits structure and behavior into two or more separate diagrams, reducing comprehension as developers are forced to flip back and forth between diagrams, attempting to integrate them mentally, which "unnecessarily strains developers' cognitive abilities." IDAR eliminates this wasteful mental effort by combining structure and behavior into one graph.

IDAR's resulting clarity should produce shorter learning curves and fewer misunderstandings and oversights, improving quality and shortening schedules.

Packages in UML may be nested, forming a hierarchy. This hierarchy, however, does not consist of roles, and the diagram inside each package is a network and not a hierarchy. Consequently, a hierarchy of packages doesn't improve understandability much. Subsystems in IDAR don't suffer from these disadvantages, and thus enjoy the full gain in understandability detailed here.

Note that organizational charts for corporations remain easy to understand regardless of their size. Because IDAR graphs are similar, they also should scale to any size and remain equally as easy to understand.

Resistance to messiness. A second important advantage of IDAR graphs over UML is they hinder the messiness (disorganization) that occurs when changes and enhancements are spliced into code with little regard for maintaining consistency of design. This claim is backed up by the following sensible constraints from the IDAR rules:

▸ The Identify rule prevents notices from initiating actions. In practice, it prevents a developer from creating spaghetti by scattering notice calls around, because notices are only allowed to convey needed information.

▸ The Down rule prevents a subordinate from commanding a superior.

▸ The Role rule prevents unexpected side effects, a common problem.

UML provides none of these defenses against messiness. For example, suppose you caused the `Modem` object in the fax machine to tell the `Receive` object to do something. This would add a line to the two UML diagrams (Figures 1 and 2) that is inconspicuous and acceptable. Doing so in the IDAR graph in Figure 4, however, would violate the Down rule because a subordinate would be commanding a superior. This is an example of the design decay that IDAR prevents.

Figure 5. Inheritance hierarchy of a CD player.

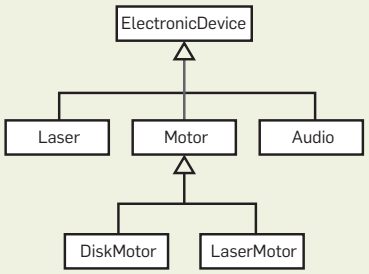
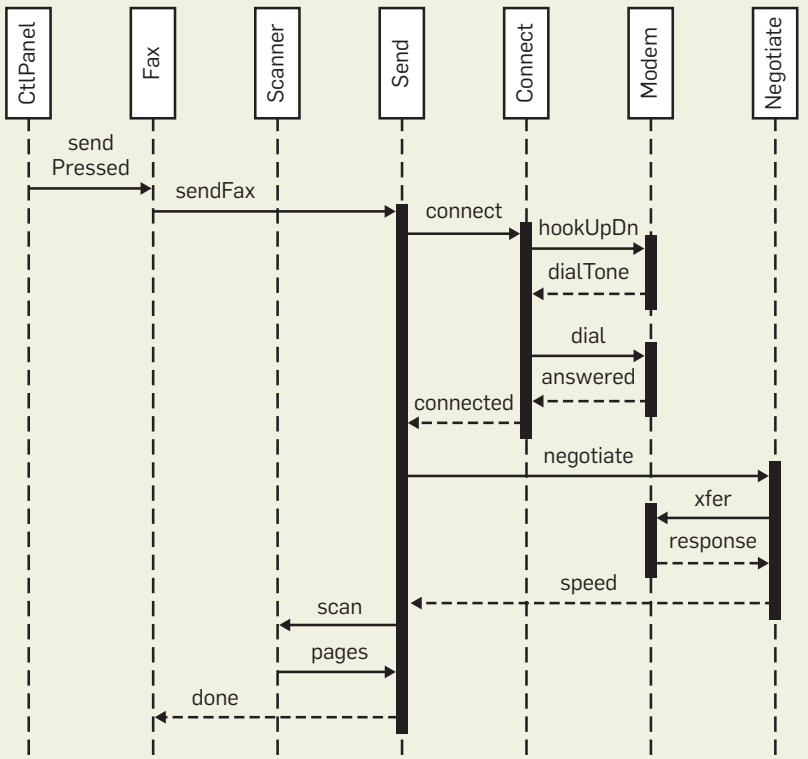


Figure 6. UML sequence diagram for sending a fax.



Limitations of IDAR Graphs

IDAR graphs do have the following limitations:

- **Object level.** IDAR is intended for object-level and subsystem-level design, so it's neither an ADL (architecture description language) nor a system modeling language. For modeling a system, OPM (Object Process Methodology)[1] is a strong contender.
- **Requires centralized control.** IDAR relies on control being organized as a command hierarchy, making it unsuitable for decentralized software with distributed control. The top levels of such software should be modeled in another way. At some level, however, the components of decentralized software are amenable to centralized control and can be designed using IDAR graphs like ordinary software.
- **Less expressive than UML.** UML can portray more views of designs than IDAR. For example, an IDAR graph is incapable of portraying transitions among states, deployment onto processors, or generalizations among classes. UML has diagrams for these and other aspects of design, and they should be employed when appropriate.

A Pilot Program

An important program was designed, coded, and deployed at Northrop Grumman using IDAR graphs. Responsible for calibration and testing of circuit boards and systems, the program is being used on the production line of an electronic product. We are forbidden from publishing this proprietary design, but we can say it has 23,000 lines of C++ code and is complex enough to have 38 classes, four subsystems, and 10 threads to handle various realtime matters. This medium-size program is not a toy.

Several people wrote and modified this program over several years, so it had become somewhat messy and was not even object oriented. The program consisted solely of tests, and I was charged with adding much nontest functionality to it, more than doubling its size. Thus, more than half of the code represents new design.

The existing code was refactored, creating objects conforming to the IDAR rules. I then designed and added the new capabilities in stages. During this process, unexpected requirements were added to the project, stress-testing the IDAR approach. IDAR graphs accomplished the following:

- Maintained clarity throughout design and implementation. Interactions among objects were so clear that any potential problems of misunderstandings among objects were avoided;
- Easily accommodated several changes and additions to the requirements. The hierarchy's clarity made it obvious where changes required by new features should be made;
- Enforced good organization;
- Did not impose excessive constraints on the design. The four rules provided enough flexibility that the design did not need to be contorted in order to satisfy them; and,
- Made design easier because the rules provided guidance. The top and bottom objects are easy to define, and defining objects between those anchor objects is not difficult. This ease of design was a surprise because imposing four rules would be expected to make designing more difficult, not easier.

Based on its results, those of us familiar with this effort believe the chief benefits of IDAR graphs over UML are their great clarity and enforcement of good organization. This pilot program was a strong success, and managers were pleased enough that they arranged for IDAR to be taught to the other software developers.

In addition to this pilot program, many trial designs have been created using IDAR graphs, and four life-size applications are described in *The IDAR Method of Software Design*.[5]

Conclusion

A hierarchy of roles appears to be essential for clearly portraying the design of any centralized organization, whether it consists of people or objects. The inability of today's object-oriented programming technology to represent this crucial kind of hierarchy is surprising, and perhaps its absence has been accepted based on the incorrect belief that an inheritance hierarchy is a suitable substitute.

An IDAR graph is clearer than UML for two main reasons: It reveals the hierarchy of roles and the breadths of those roles; and the triads (why-what-how) offer deeper insights into the nature of objects. UML cannot provide these. Given that IDAR graphs are clearer than UML, and that the four rules underlying them resist messiness, developers should produce fewer bugs when designing and implementing software using IDAR graphs. The result will be improved quality and timeliness.

This article contains enough information to enable readers to create designs using IDAR graphs. For more information, you can download the slides from a presentation at the IEEE Software Technology Conference in 2014.[4] Also, refer to *The IDAR Method of Software Design*,[5] which not only details this method (and related topics), but also includes the four life-size applications mentioned in this article.

Acknowledgments. I am indebted to Jim Ray for his consistent support of the IDAR method. Thanks to Jim Wilk and Dorothy Kennedy for their unswerving support. Sammy Messian managed the pilot program that used IDAR and values it for the fine results it produced. All four are managers at Northrop Grumman.

Related articles on queue.acm.org

UML Fever: Diagnosis and Recovery
Alex E. Bell
http://queue.acm.org/detail.cfm?id=1053347

Coding for the Code
Friedrich Steimann and Thomas Kühne
http://queue.acm.org/detail.cfm?id=1113336

Software Development with Code Maps
Robert DeLine, Gina Venolia, and Kael Rowan
http://queue.acm.org/detail.cfm?id=1831329

References

1. Dori, D. Why significant UML change is unlikely. *Commun. ACM 45*, 11 (2002), 82–85.
2. Kiczales, G., et al. Aspect-oriented programming. *Proceedings of the 11th European Conference on Object-Oriented Programming*: (1997), 220–242.
3. Moody, D., van Hillegersberg, J. Evaluating the visual syntax of UML: An analysis of the cognitive effectiveness of the UML family of diagrams. *Software Language Engineering.* Springer-Verlag, Berlin, Heidelberg, 2009, 16–34.
4. Overton, M. Command graphs: a significant improvement to OOP/OOD. *IEEE Software Technology Conference*, 2014; http://conferences.computer.org/stc/2014/index.html or http://www.ieee-stc.org.
5. Overton, M. 2014. *The IDAR Method of Software Design.* CreateSpace, Seattle, WA, 2014.
6. Rumbaugh, J., Jacobson, I., Booch, G. *Unified Modeling Language Reference Manual*, 2nd ed. Addison-Wesley, Boston, MA, 2004.
7. Vicente, K. *Cognitive Work Analysis.* Lawrence Erlbaum Associates, Mahway, NJ, 1999, 174–176.

Mark Overton is a software engineer at Northrop Grumman working on various parts of software-defined radios. He previously worked at HP, contributing to both the architecture and implementation of all-in-one printers, particularly their scanners.