Combining programming by example and real-world analogies, users create new behavior out of existing behavior.

Programming by Analogous Examples

Why do end users need to program? Anyone can become overwhelmed trying to cope with information in a world flooded by information. In light of the ubiquity of computer networks, the information challenge for end users is no longer about accessing information but about processing it. The direct-manipulation paradigm, popularized in the 1980s, begins to break down when they are unable to directly manipulate the sources or their information, such as the location of their files on a hard drive or the emails in their in-boxes.

End-user programming is emerging as a crucial instrument in the daily information-processing struggle [8]. Such programming allows customized personal information processing. But few end users have the background, motivation, or time to use traditional programming approaches or the means to hire professional programmers to create programs for them. Simple forms of end-user programming include email filters to clean up email by directing specified emails into separate folders and spreadsheets to explore, say, the total cost of a new house.

Programming by example (PBE) is a powerful end-user-programming paradigm enabling users without formal training in programming to create sophisticated programs. PBE environments create programs for end users by observing and recording software behaviors as they manipulate information on the graphical user interface (GUI) level. For example, in Microsoft Word, PBE mechanisms allow users to build macros.

Here, we explore the problem of PBE reuse, or how users can reuse old program behavior. A user might find a PBE-generated program useful but may need to either generalize or modify it for a related but different task. Program reuse is a well-known software
design problem. Reuse problems hamper the productivity of a wide range of software projects—from individual end-user programmers to large distributed teams of software developers. In the PBE context, reuse poses even more complex issues. While PBE initially shields end users from having to deal with programming issues, reuse forces them to leap cognitively between two levels of representation:

**The GUI level.** This level features familiar windows, icons, and menus. For instance, in a word processing application, such as Word, this level represents content directly manipulated by users with such operations as the typing of new text, the formatting of text, and the use of cursor keys to navigate through a document.

**The program level.** This level captures user manipulations directly into programs so users can replay them. In Word, this level incorporates Visual Basic. User manipulations are recorded as Visual Basic scripts; users then assign scripts to keyboard commands or to user-defined toolbar commands.

The PBE representation chasm mirrors the difficulty users have comprehending the mapping between the GUI level and the program level. In reuse, this chasm is especially problematic, because users are required to have at least a minimal understanding of the program-level representations—if they want to be able to adapt these representations to their needs. For end users with no previous exposure to programming in Visual Basic or even to programming in general, this transition may be too complex and leave them frustrated. Frustrated, they may give up on the idea of end-user programming and continue to solve their original problems manually over and over again for years.

Analogies are powerful cognitive mechanisms people use to construct new knowledge from knowledge they’ve already acquired and understand. When analogies are used with PBE, the result is a new end-user programming paradigm combining the elegance of PBE to create programs and the power of analogies to reuse existing programs. We call the combination of PBE and analogies “programming by analogous examples.” Analogies are an effective representation level bridging the GUI and program levels.

Two detailed examples help us explore the reuse problem. In the first, we created a Word macro for repetitive reformatting in order to understand macro reuse. We chose this particular example not because the Word macro recording mechanism is the most sophisticated PBE system, but because Word has an enormous user base, and readers may be able to relate more read-
ily to use and reuse issues in the Word context. In
the second example, we built a SimCity-like simula-
tion using the AgentSheets simulation authoring
this tool provides an end-user programming
approach considerably more accessible than Visual
Basic. We’ve found that even when the program
level is relatively accessible, analogies are still needed
as a PBE reuse mechanism.

As PBE techniques become more widely adopted
in both commercial and research systems, end users
incrementally construct and edit new behavior and
commands as needed. As in conventional program-
ming approaches, the effective reuse of new behavior
is limited by the underlying representations. These rep-
resentations run the risk of presenting end-user pro-
grammers with the same copy-paste-modify
and inherit-reuse problems that pro-
fessional pro-
grammers have
struggled with for
years. What makes reuse even more problematic in a
PBE situation is that the programs that need to be
modified for reuse are machine generated, typically
containing little information as to how they might
be changed.

The reuse scenario in Figure 1 involves an end
user trying to automatically reformat an address list.
Word and other applications allow for modular
modification through a macro mechanism. Users are
given an interface to program macros by PBE
through a menu command called “Record New
Macro.” Additional support is given by the macro-
editing toolbar and menu functions. Once a macro
is named and assigned to a toolbar icon or to a key-
board command, it is created simply by performing
the desired task—repetitive reformatting in our
example. The system records keystrokes and com-
piles the command into Visual Basic. The command
can then be invoked by anyone editing a document.

End users want to modify regularly and, hence,
reuse a macro with additional, related actions, then
assign the entire new set of changes to the same key-
board command. Mechanisms are built into Word
for renaming the macro, thus copying the same
behavior, but any modification to an existing macro
must be done through the Visual Basic program-
ning language. For instance, how would a user have
to modify the macro to reformat not just the
Fax: but also the Address: fields in Figure 1.
To modify this macro through the “Record” function,
a user would have to overwrite the old macro,
go through the same actions that were recorded ear-
lier, and then record the desired additional behavior.

To be able to reuse existing macros, making them
useful in more general situations, end users have to
drop from the GUI level (Figure 1, left) to the pro-
gram level (Figure 1, right). According to
Microsoft’s own Word documentation: “Recorded
macros are great when you want to perform exactly
the same task every time you run the macro. But
what if you want to automate a task in which actions
vary with the situation, or depend on user input … ? To create
powerful automations, you should learn to
program in Visual
Basic for Applica-
tions.”

There is a large cog-
nitive chasm between
the two levels. The
GUI one provides only
limited options for
reuse. Even though advanced reuse requires
making a transition to the program level, no inter-
mediate steps are provided to take the user from the
simple recording mechanism to the Visual Basic
programming language. At the GUI level, users are
limited in reuse to copying macros as black boxes
into other documents or to making macros gener-
ally available to all documents. At the program
level, code may be copied, pasted, and modified
with the same difficulty as these commands are
invoked in any other object-oriented program. At
this point, the user has jumped from the comfort-
able environment of direct manipulation at the
GUI level directly into the uncomfortable or down-
right frustrating program level.

How Analogies Bridge the Chasm
The AgentSheets simulation authoring tool helps
explain how analogies in a PBE system can bridge
the chasm between the GUI level and the program
level. In AgentSheets, end users can build SimCity-
like simulations and export them as interactive Java
applets. (For a detailed description of AgentSheets,
see www.agentsheets.com/userforum-publications.
html.)

AgentSheets combines PBE with graphical
rewrite rules into an end-user programming para-
Graphical rewrite rules (GRRs), which are also used in such systems as Cocoa/KidSim and Creator, are powerful languages for expressing the concept of change in a visual representation. GRRs declaratively describe spatial transformations with a sequence of two or more dimensional situations containing objects. Situations can be interpreted with respect to objects contained in them and the spatial relationships among these objects. The differences among situations imply one or more actions capable of transforming one situation into another. In AgentSheets, any number of GRRs can be aggregated to create complex behaviors for agents; these agents and their behaviors combine and interact to simulate any scenario a user can imagine—from heat diffusion to urban planning. We have found it likely that behaviors created in one simulation are valuable to users building other simulations (see Figure 2). Compared to the GUI-program chasm in Word, the GUI-program chasm in AgentSheets is much less daunting for end users. The GRR representation at the program level closely matches the representation at the GUI level. For instance, the user-produced train and train-track icons at the GUI level are also found at the program level. This representational mapping helps end users comprehend programs. But despite the closer match, it is still necessary for any end-user-programmable application to have a mechanism for dealing with reuse.

**An Analogy (Making Cars Move Like Trains)**

In an application called Sustainopolis used by urban planners to explore public transportation issues, an end user would want to incorporate cars and streets. Noticing that trains and tracks are already programmed and realizing that cars move on streets much like trains move on tracks, the user would want to reuse the “move” behavior already written for the train object and attach it to the car object.

One approach would be for the user to start from scratch and simply demonstrate all the examples of car and street interactions. Unfortunately, the set of rules attached to the train is quite large. The problem is that even for a relatively simple behavior, such as making the train follow the track, rules have to be demonstrated to specify all the meaningful combinations of interactions between trains and tracks. Trains are capable of heading in four different directions—north, south, east, and west. A complete specification accounting for 15 different train-track pieces—straight; four orientations of curves; four different T intersections and one crossing; and four orientations of cul-de-sacs—requires 256 rules.

These rules specify behavior in various situations, including: What if the train drives into a cul-de-sac? and Which way should the train go if it has a choice of going left or right? If the tracks are arranged care-
fully in the simulation, a good number of rules can be eliminated, but the point remains that it must have taken quite some time and patience to demonstrate the complete set of rules. Given that this behavior exists already in our scenario, it would be preferable to find a way to avoid having to demonstrate the entire set of rules again to specify the conceptually similar interaction between cars and streets.

A different approach is to copy all the train's rules into the car and manually edit them by substituting corresponding icons. In most GRR-based systems, the user would then copy all the rules from trains and tracks to streets and cars, then simply substitute cars for trains and streets for tracks. But transferring rules from one application scenario to another is basically the same arduous, error-prone process a programmer goes through to reuse a program or a Word user goes through to reuse a macro. So, is the user's time being saved? Without understanding what the user intends, how can the system facilitate the reuse process? We have found that the mechanism allowing an end user to program allows reuse as well.

Programming by analogous examples supports the reuse of previously recorded example programs. Instead of creating car behavior from scratch—either through demonstration or through manual editing—users generate a complete set of rules by specifying an analogy.

It is the relationship between trains and tracks that the user wishes to reuse for cars and streets. In contrast to object-oriented programming in which inheritance is used to define an ontology of object classes, analogies are used to define an ontology of object relationships represented by common active verbs, such as “move” (see Figure 3).

The user defines analogies through the analogy dialogue box (see Figure 4). The result is that the behavior programmed through GRRs for trains moving on tracks is transferred to cars moving on streets. The verb “moves” is differentiated for trains and cars, so trains do not move on streets, nor do cars move on tracks, because no relationship is implied between trains and streets, or between cars and tracks.

Here, reuse is expressed in terms of existing objects, and abstractions are not required. The result of the analogy is a new set of GRRs attached to cars, allowing them to move on streets. All the rules generated by analogy are like ordinary GRRs in that the end user edits them to deal with exceptions to the analogy.

Fortunately, PBE systems often feature representations that—through the use of analogies—serve as a bridge between the GUI level and the program level. In the train-car analogy, the end user created new behavior out of existing behavior. This method employs reuse-enhanced programming in which an initial set of programs is created through PBE and then extended through analogies.

At what depth does an analogy mechanism need to “understand” a program to create an analogous pro-
gram? Mere syntactic symbol substitution is insufficient for the transformation of nontrivial programs. A deep understanding of the program semantic is not only an extraordinarily difficult and unsolved AI problem, but developing analogies would probably require users to annotate programs extensively with semantic information.

From Substitutions to Analogies

In the analogy specified in Figure 3, the four icons are from a much larger set of icons representing cars and trains with different headings, as well as streets and tracks representing different topologies. A mechanism that would merely substitute the car and street icons for the train and track icons in order to create a set of analogous rewrite rules would not be very useful; it would match only a limited subset of all relevant rules. How does the analogy system establish the more general mapping among all these different but related items?

Analogy mechanisms cannot stop at syntactic substitution but need access to some minimal semantic information in order to establish more general correspondence. It is no trivial matter for a computer system to substitute correctly at all the necessary levels to render the resulting code usable without further editing by a human. More than 10 years ago, Clayton Lewis at the University of Colorado in Boulder proposed a concept he called “pupstitution” to decrease the brittleness inherent in straight copy-and-paste techniques [5].

In the AgentSheets substrate, “systematicity,” or correspondence, in analogy is facilitated through structural and behavioral similarity between the base and target objects [9]. Analogies are made by specifying which relationship should be transferred. For example, cars and trains are objects that can be as simple or as complex as a user wants and can include a single applicable behavior or many behaviors. However, the user has to specify that cars move like trains to transfer this specific behavior. And for the transfer to occur with a high degree of systematicity (and effectiveness to the user), the street and track objects must be structurally similar.

AgentSheets assures this high-fidelity transfer with connectivity information. From the base icons created for the urban-planning icon gallery, the system applies syntactic transformations at the user’s request, automatically creating meaningful variations to illustrate intersections and curves, as well as directional orientation [10]. Syntactic transformations create the visual variations necessary for placing the agents on the AgentSheets worksheet and saves the user hours of work on icon depictions. Icons are then annotated by the user with semantic information, such as connectivity (see Figure 5). Connectivity defines the input and output ports for each icon. For instance, a horizontal street segment icon has inputs/outputs to its left and right. Transforming the annotated track creates an entire family of track icons that act essentially as throughputs for objects moving on them.

The system uses an icon’s semantic and syntactic connectivity information to support pupstitution in
programming by analogous examples. By specifying the connectivity of an icon, then transforming it, the “move” GRR acquires a useful dimension of complexity. After the analogy is established, cars have a rule set that is isomorphic to the train rule set, that is, cars know how to move on all kinds of streets. The system uses the semantic information available to ensure high systematicity between the base and the target; thus, the correct mapping between tracks and streets is established without the end user’s intervention.

The general insight we derived is that a little semantics is necessary for creating meaningful analogies. This means that end users have to provide some additional up-front information to annotate their designs with minimal semantic annotation. It also means that this kind of semantic annotation dramatically improves the reusability of behavior. For AgentSheets, users need to provide only semantic information to the base agents, specifying, say, the connectivity of a street agent they have already defined. AgentSheets can automatically transform agents representing flow conductors, including streets, wires, and pipes—syntactically (by applying geometric transformations to the agent bitmap) and semantically (by applying topological transformations to the agent’s connectivity).

**Reuse through Inheritance**

In object-oriented programming, in order to facilitate reuse, inheritance can serve the role of the reuse mechanism in PBE systems. An object subclass not only inherits the behavior of the superclass but can overwrite and extend that behavior. How would inheritance work in our trains and cars example? One easy way to get analogous behavior is to make “car” a type of, or subclass of, “train,” and “street” a subclass of “track.” While this approach produces the desired behavior (due to inheritance), it is ontologically unsound, and changing the object hierarchy in this way produces a misleading model based on weak design.

To correct this problem, system developers expect end users to be a bit more like programmers. Therefore, in the class hierarchy, cars become siblings of trains and streets siblings of tracks through two abstract superclasses—“moving-object” and “movement-guiding-object.”

For the user to get the trains and cars to move, a generalized GRR has to be expressed in terms of “moving object” and “movement-guiding-object.” However, while this new class hierarchy may be ontologically sound, it also introduces a couple of problems:

**Need for abstractions.** Abstractions for end users are nontrivial and difficult to represent visually, especially when objects are represented by user-defined icons. How would abstract objects embodying moving objects and movement-guiding objects look, and who would have to draw them?

**Overgeneralization.** If city traffic is run with this new representation in place, trains can move on streets and cars can move on tracks; although this scenario is a theoretical possibility, it should not be allowed to happen in the urban-planning-domain application. In order to get around this real-life constraint, the behavior of trains and cars would have to be specialized again to prevent such unwanted combinations of moving objects and movement-guiding objects.

**Conclusion**

In PBE systems, the representation end users see at the GUI level may differ greatly from the representation at the program level. In these systems, special representations are often used to ease the end user’s transition between the two levels. These representational features enable programming by analogous examples, simplifying program reuse.

Our work with programming by analogous examples is at an early but promising stage. The combination of some semantic information with structural information has allowed the reuse of complex behaviors in the context of interactive simulations. The more similarity there is between behavior the user has created and wants to reuse and the target behavior, the better is the analogy and the less likely the need for editing. We seek to increase differentiation by end users trying to determine appropriate analogical matches. For instance, cars moving on
streets and electricity moving through wires share similarities, but electricity can go in both directions at an intersection. Therefore, an analogy between these two classes of object would produce crash-prone cars. We'll soon also explore the use of programming by analogous examples in nonsimulation application domains.

References

Alexander Repenning (ralex@cs.colorado.edu) is CEO of AgentSheets, Inc., and a professor in the Department of Computer Science and the Center of LifeLong Learning & Design at the University of Colorado in Boulder.

Corrina Perrone (corrina@agentsheets) is a project manager in AgentSheets, Inc., and director of media relations for an international nonprofit organization.

This research is supported by the National Science Foundation under grants: DMI-9761360, RED 925-3425, Supplement to RED 925-3425, REC 9804930, REC-9631396, and CDA-940860.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.