Graphical Rewrite Rule Analogies: Avoiding the Inherit or Copy & Paste Reuse Dilemma

Corrina Perrone and Alexander Repenning

Center of LifeLong Learning & Design
Campus Box 430
University of Colorado, Boulder CO 80309
(303) 492-1349, {ralex, corrina}@cs.colorado.edu
Fax: (303) 492-2844

ABSTRACT

Reuse mechanisms, such as inheritance in an object-oriented programming approach, are useful to professional programmers but fail to support the occasional programming needs of the end-user. Consequently, a surprisingly high percentage of end-users resort to "copy and paste" approaches for reuse instead of making appropriate use of object-oriented techniques. Visual Analogies are a reuse mechanism for end-users who otherwise would have resorted to "copy and paste." This paper illustrates how visual analogies avoid some of the problems intrinsic to object-oriented programming by eliminating the pitfall of overgeneralization and the need to create non-concrete programming abstractions.

THE REUSE DILEMMA: INHERIT OR COPY & PASTE?

From an end-user perspective, software reuse takes several forms. Historically, the behavior of most application programs could not be changed or augmented without substantial reprogramming [7]. As techniques emerge to support end-user programming and end-user modifiability, it becomes clear that end-users have little interest in programming computers unless it will help them to build tools that enhance their domain productivity [17].

Therefore, the inclusion of mechanisms to aid the end-user in the location and modification of code that performs a function similar to the one desired greatly improves the usability of domain applications.

By allowing incremental development, object-oriented languages attempt to provide reuse mechanisms such as inheritance. End-users tend to be better at thinking concretely than abstractly [17], and for this reason, inheritance works for professional programmers trained in abstraction processes, but fails to work for end-users.

Ideally in an object oriented system, the user locates a promising class in the object class hierarchy and refines it by adding or extending methods to provide the new desired behavior. Few situations approach this ideal, however, and the result is that the user is faced with altering the object hierarchy. It is here that the process breaks down in the face of an inheritance structure defined by someone else. Some behavior of the original objects is desirable, other behavior might be merely superfluous, and still other behavior might be undesirable.

Several reactions to this phenomenon have emerged. By the object oriented programming community, end-users are often dismissed or forgotten as they grapple unsuccessfully with the complexities of inheritance. Other applications, such as Hypercard, avoid the situation by resorting to inheritance-free representations. For many end-user programmers, this is appealing, although it is a lowest common denominator approach. This "less is more" philosophy of HyperCard has enabled a surprising number of end-users to program. Reuse in HyperCard consists of copy and paste. In this approach, the potentially useful connection between the source and copy is lost.

In this paper, we introduce a new reuse approach called Graphical Rewrite Rule Analogies (GRRAs) that are built into the Agentsheets programming substrate [18,19, 20, 21], We contrast Graphical Rewrite Rule Analogies with an object-oriented approach using inheritance in the context of a reuse scenario. Analogies are not limited to graphical rewrite rules and have been explored in AgentSheets most recent programming language called Visual AgentTalk [3]. Finally, we discuss the applicability of GRRAs in the face of analogical brittleness and achievable end-user modifiability.

SIMCITY IN 10 MINUTES: A REUSE SCENARIO

Agentsheets is a Macintosh programming substrate for creating domain-oriented visual environments. In the last four years, Agentsheets has been used to create more than 40 educational and industrial applications serving as construction kits, simulation environments, visual programming languages, design environments, and games.

We use a scenario to explain the issues arising from reuse. In an application called CityTraffic (Figure 1) used to model traffic patterns, an end-user, in this case a urban planner, wishes to incorporate cars and roads. Noticing that Trains and Tracks are already successfully programmed, and realizing that cars move on roads similar to the way Train moves on Track, the user wishes to reuse the *move* behavior already written for the Train object and attach it to the Car object.

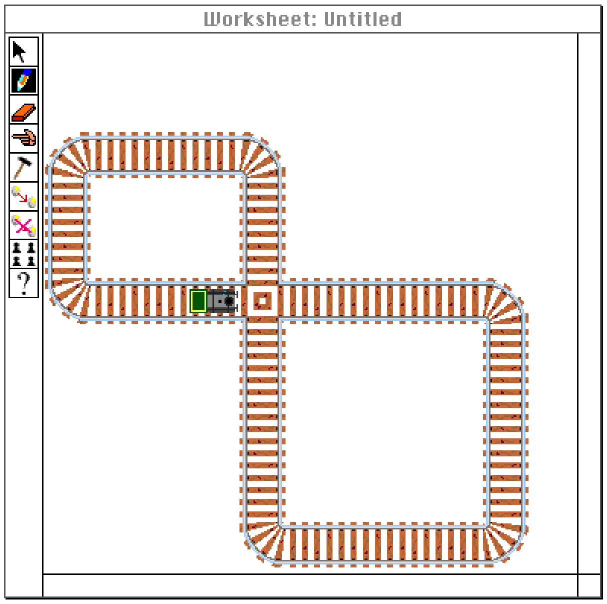


Figure 1. CityTraffic: Trains move on Tracks

The following two sections discuss and contrast reuse approaches in the context of the CityTraffic application.

Using Inheritance

One easy way to get this behavior is to make Car a type of, or sub-class of Train, and Road a sub-class of Track, as in figure 2. While this approach will produce the desired behavior because of inheritance, it is ontologically unsound, and changing the object hierarchy in this way produces a misleading model based on weak design justifications.

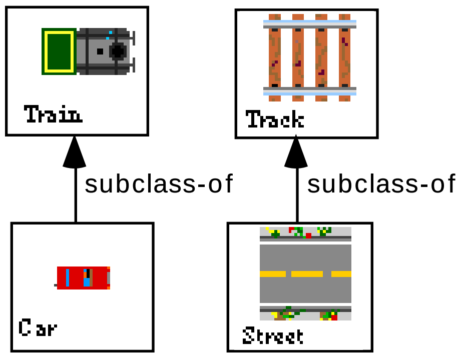


Figure 2. Acquiring Behavior by Inheritance

In order to correct this problem, it is expected that the end-user become a bit more of a programmer, which in itself can be an erroneous expectation. In the class hierarchy Cars can become siblings of Trains and Roads sibling of Tracks by creating two abstract super classes: Moving-Object and Movement-Guiding-Object (Figure 3).

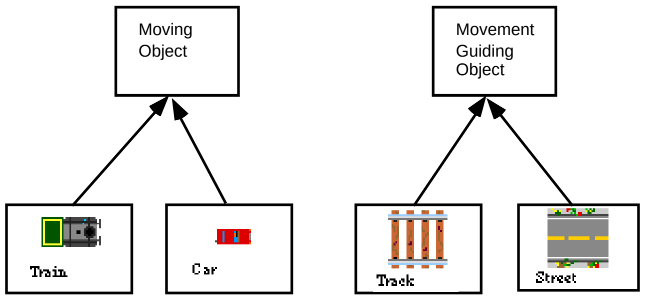


Figure 3. Creating Siblings

Program fragments guiding the behavior of the Train moving on Tracks need to be *generalized* in order to also enable Cars to move on Roads. The Train behavior was previously expressed with graphical rewrite rules [1,9,13,18,22] (Figure 4).

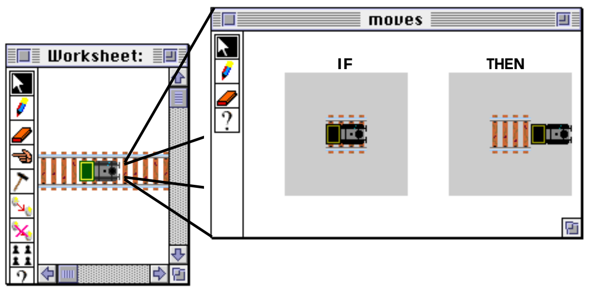


Figure 4. A Graphical Rewrite Rule

To generalize these rules means to express the rules in terms of the abstract classes. Specifically, Trains need to be substituted with Moving-Object and Tracks with Movement-Guiding-Object (Figure 5):

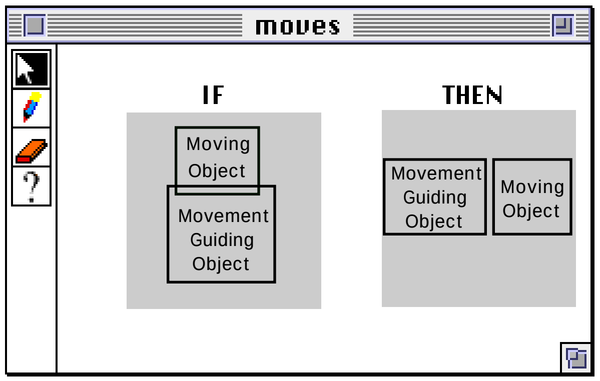


Figure 5. Abstract Rewrite Rule

While this new class hierarchy may be ontologically sound, it introduces two very serious problems:

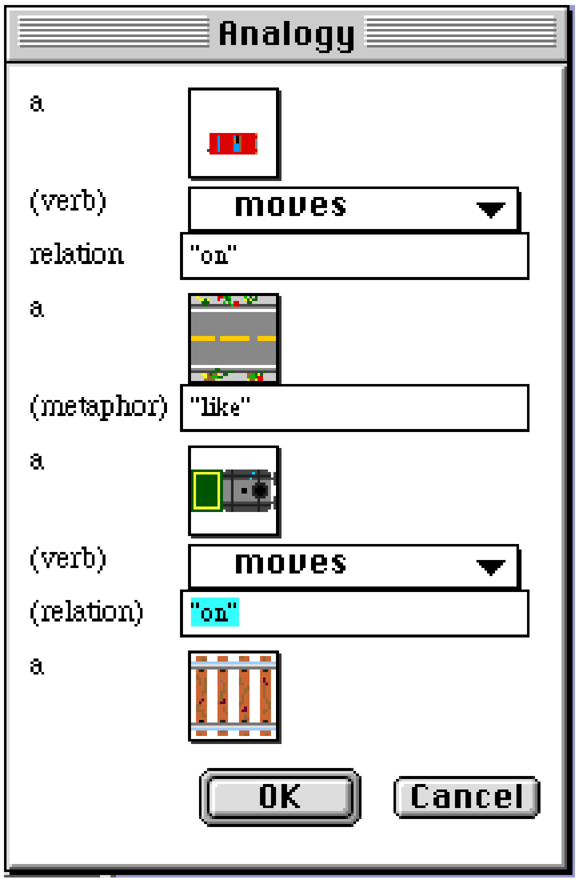
- ***Overgeneralization***. If city traffic is run with this new representation in place, Trains can now move on Roads and Cars will now move on Tracks, which although theoretically possible, should not be allowed to happen in the urban planning domain application. In order to get around this real-life constraint, the behavior of Trains and Cars would need to be specialized again to prevent these unwanted combinations of Moving-Objects and Movement-Guiding-Objects.
- ***Non-Visual Abstractions***. Another problem with this new representation is especially apparent in the visual arena. Abstractions are hard to represent in visual programming approaches. Urban planners have little use for abstract classes that have no intuitive visualization such as Moving Object and Movement Guiding Object. These super-classes are unintuitive non-visual abstractions. When another end-user wishes to add Trucks to the model, the true object hierarchy must be retrieved, and the end-user must again foray into the programmer's realm, this time understanding not only that Truck is a Moving Object, and that its move behavior must be specialized to preclude it from moving on Tracks, but also that Trucks have a visual depiction that is not inherited.

Using Graphical Rewrite Rule Analogies

A very different approach to solve the problem is the use of Graphical Rewrite Rule Analogies. What the user wishes to communicate to the system is:

Cars move on Roads
like
Trains move on Tracks

By using Graphical Rewrite Rule Analogies within Agentsheets, the user can specify this by way of an analogy dialog box (Figure 6). The result is that the behavior programmed via graphical rewrite rule for Train moves on Track is transferred to a Car object on Road. The verb moves is differentiated for Trains and Cars, so that Trains do not now move on Roads nor will Cars move on Tracks.


Analogy

a

(verb) moves

relation "on"

a

(metaphor) "like"

a

(verb) moves

(relation) "on"

a

Figure 6. Making an Analogy

The result of the analogy is a new rewrite rule attached to cars allowing cars to move on roads.

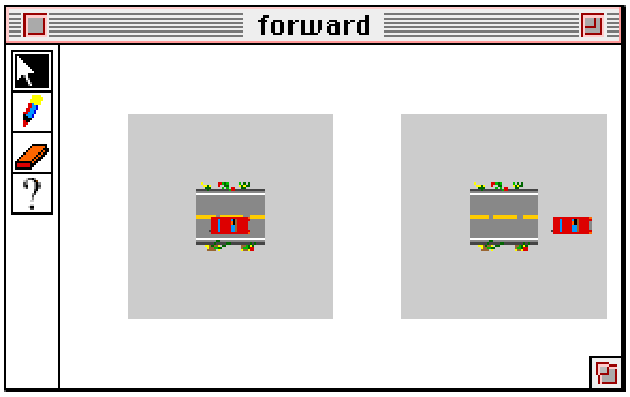


Figure 7. Rule created by Analogy

Gentner and others [4, 10,15,16] recognize several critical aspects of analogies, including clarity, richness, and systematically, where systematically is derived from structure mapping theory. Structure mapping theory distinguishes between attributional similarity of objects and relational similarity of objects. Relational similarity allows for first-order relations, which take objects as arguments and higher-order relations, which take propositions as arguments. High systematically in an analogy means that many higher-order relations are shared between the base and target domains.

Reuse of code in the object oriented programming paradigm [2, 11, 23] depends on a systematically match between the behavior of the original code and the behavior of the target code. The most efficient reuse of object oriented code can occur when there is both a great deal of structural similarity and systematically between the base and target programming tasks. When seeking to reuse code, an end-user seeks a good analogy, i.e. a clear, rich and systematic mapping between the code sought for reuse, and the envisioned new application code. The more analogous the base code, the fewer modifications required for reuse, which implies a decreased necessity for the end-user attempting to program to understand *any* programming paradigm.

Graphical Rewrite Rule Analogies can be used to increase the analogous match between the base code and the envisioned target code. This allows the user to reuse precisely the behavior that is desired avoiding overgeneralization the need for non-visual abstractions.

APPLICABILITY OF GRAPHICAL REWRITE RULE ANALOGIES

Graphical Rewrite Rule Analogies provide an alternative programming solution which facilitates reuse and avoids the discussed pitfalls of an object oriented approach. Questions remain to be answered. Lewis [12] have brought up interesting questions about the robustness of automatic cut and paste techniques. Simple substitution in these cases is often over-specific and can add annoying steps to the process for little gain. It is no trivial matter for a computer system to substitute correctly *at all the necessary levels* required to render the resulting code useful and usable without further editing by a human. Lewis proposes a concept called *pupstitution* to decrease the brittleness inherent in straight copy and paste techniques.

In the Agentsheets substrate, related concepts are represented by related icons. From the base icons created for the gallery, syntactic transformations can be applied to automatically create meaningful variations to illustrate intersections and curves, as well as directional orientation [21]. Icons may also be annotated with semantic information such as connectivity (Figure 8), which gives the machine that displays them information about what they mean.

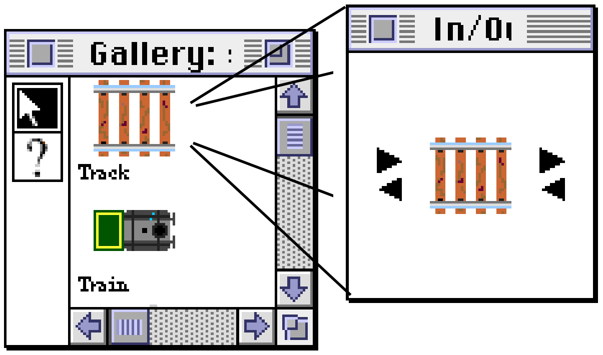


Figure 8. Annotating Icon with Connectivity Semantics

Transforming the annotated Track creates an entire family of Track icons (Figure 9).

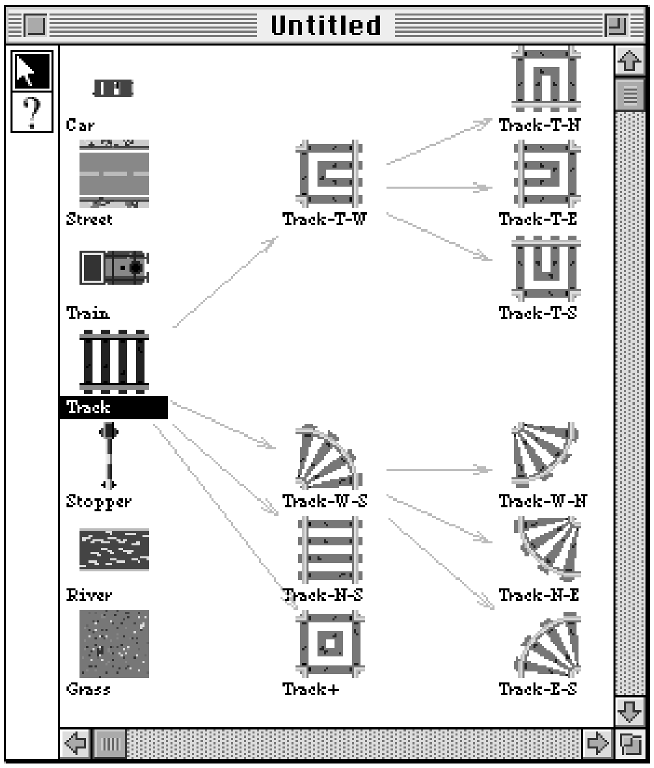


Figure 9. Transformed Tracks

This semantic and syntactic connectivity information about an icon can be used to support pupstitution in analogous cut and paste. If we apply Graphical Rewrite Rule Analogies to our example as described in the previous section, we achieve the desired behavior in the Car object, however, we are not quite to the point where a Car object (or a Train object, for that matter) can move on all types of appropriate surfaces. Straight Tracks and straight Roads will be handled correctly, but at an intersection or a curve, neither object will know how to behave given the move rule we have defined for it. To provide this ability, it will be necessary to define curve and intersection Track icons, and then specify rules of movement for Trains on each of these types of Tracks. Transforming these rules to Car by analogy still requires the end-user programmer to map the appropriate type of Track to the corresponding Road icon. By specifying the connectivity of an icon (as in Figure 8), and then transforming it, the *move* rule acquires a useful dimension of complexity. Trains now know how to move on all kinds of Tracks. When an analogy to this situation is applied, the system uses this semantic information and the correct mapping between Tracks and Roads can be made without the user's intervention.

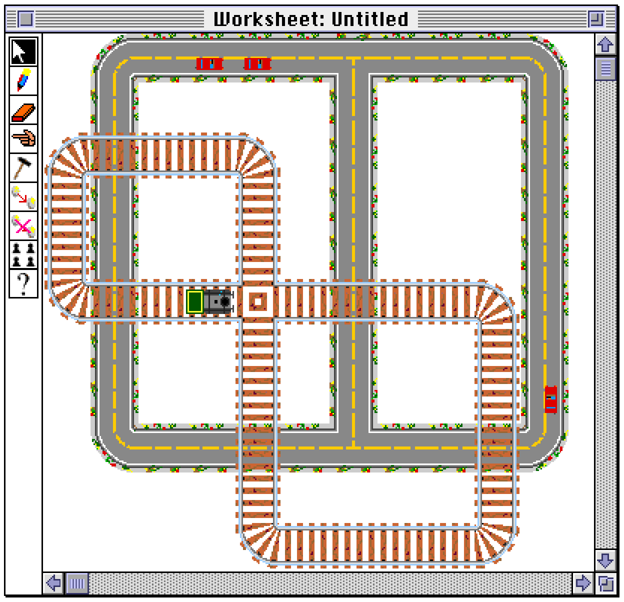


Figure 7. Planning City Traffic

CONCLUSION

It is highly desirable for end-users to modify or enhance domain applications, and mechanisms are needed to provide this flexibly without forcing end-users to become programmers. Object-oriented concepts relying on single and multiple inheritance can be too complex for end-users, and non-inheritance approaches too limited. Graphical Rewrite Rule Analogies allow reuse without the oversimplification, overgeneralization and unnecessary abstraction pitfalls of these other approaches.

ACKNOWLEDGMENTS

The authors wish to acknowledge Clayton Lewis, and the members of the Human Computer Communication group at the University of Colorado. The research was supported by the National Science Foundation under grants No. DMI-9761360, RED 925-3425, and Supplement to RED 925-3425.

REFERENCES

1. Bell, B., “ChemTrains: A Visual Programming Language for Building Simulations,” Technical Report, CU-CS-529-91, Department of Computer Science, University of Colorado at Boulder, Boulder, Colorado, 1991.

2. Booch, Grady, *Object Oriented Design with Applications*, Benjamin/Cummings, Reading, MA, 1994.

3. Craig, B., "Behavior Combination Through Analogy," *Proceedings of the 1997 IEEE International Symposium on Visual Languages*, Capri, Italy, IEEEE Computer Society, 1997, pp. 270-273.

4. Chee, Y.S., "Applying Gentner's Theory of Analogy to the Teaching of Computer Programming", *Int. Journal of Man Machine Studies*, Vol. 38, pp. 347 - 368, 1993.

5. Clement, C.A., and Gentner, D., "Systematicity as a Selection Constraint in Analogical Mapping", *Cognitive Science*, Vol. 15, pp. 89-132, 1991.

6. Dershowitz, N., "Programming by Analogy", in *Machine Learning: An Artificial Intelligence Approach, Volume II*, R.S. Michalski, J.G. Carbonell, T.M. Mitchell, eds., Morgan Kaufmann Publishers, Los Altos, CA, 1986, pp. 395 -423, ch. 15.

7. Eisenberg, M., and Fischer, G., *"Programmable Design Environments: Integrating End-User Programming with Domain-Oriented Assistance,"* CHI '94, ACM Press, 1994, pp. 431-437.

8. Fischer, G., Henninger, S., and Redmiles D., "Cognitive Tools for Locating and Comprehending Software Objects for Reuse," IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 318-328.

9. Furnas, G. W., “New Graphical Reasoning Models for Understanding Graphical Interfaces,” *Proceedings CHI’91*, New Orleans, LA, 1991, pp. 71-78.

10 Goldstone, R.L., Medin, D.L., and Gentner, D., "Relational Similarity and the Nonindependence of Features in Similarity Judgments", *Cognitive Psychology*, Vol.23, pp. 222-262, 1991.

11. Jacobsen, Ivar, et. al., *Object Oriented Software Engineering - A Use Case Driven Approach*, Addison-Wesley, 1993.

12. Lewis, C., "Some Learnability Results for Analogical Generalization", Technical Report CU-CS-384-88, University of Colorado, 1988.

13. Lieberman, H., “An Example-Based Environment for Beginning Programmers,” in *Artificial Intelligence and Education*, R. W. Lawler and M. Yazdani, Ed., Ablex Publishing, Norwood, NJ, 1987, pp. 135-151.

14. Majidi, M., and Redmiles, D. "A Knowledged-Based Interface to Promote Software Understanding", *Proceedings of the 6th Annual Knowledge-Based Software Engineering Conference*, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 178 - 185.

15. Medin, D.L., Goldstone, R.L., and Gentner, D., "Respects for Similarity", Psychological Review, Vol. 100, No. 2, pp. 254-278, 1993.

16. Medin, D.L., Goldstone, R.L., and Gentner, D., "Similarity Involving Attributes and Relations: Judgments of Similarity and Difference are not Inverses", *Psychological Science*, Vol. 1, No. 1, 1990.

17. Nardi, B.A., *A Small Matter of Programming: Perspectives on End-User Computing*, The M.I.T Press, Cambridge, MA, 1993.

18. Repenning, A., “Agentsheets: A Tool for Building Domain-Oriented Visual Programming Environments,” Ph.D. dissertation, Dept. of Computer Science, University of Colorado, 1993.

19. Repenning, A., “Agentsheets: A Tool for Building Domain-Oriented Visual Programming Environments,” *INTERCHI '93, Conference on Human Factors in Computing Systems*, Amsterdam, NL, 1993, pp. 142-143.

20. Repenning, A., “Designing Domain-Oriented Visual End User Programming Environments,” *submitted to: Journal of Interactive Learning Environments, Special Issue on End-User Environments*, , pp. 1994.

21. Repenning, A., "Bending Icons: Syntactic and Semantic Transformation of Icons", *to appear in: Visual Languages '94.*

22. Smith, D. C., A. Cypher and J. Spohrer, “KidSim: Programming Agents Without a Programming Language,” *Communications of the ACM*, Vol. 37, pp. 54-68, 1994

23. Wilkie, George, *Object-Oriented Software Engineering*, Addison-Wesley, 1994.