Composition not Inheritance: Imagining a Functional Digital Humanities

Andrew Pilsch

Today, I want to talk about software architecture in some detail and how such a fine-grained consideration of industrial programming practices can suggest new, interesting, and I would add important directions for thinking about how we do our business in the digital humanities. As background, I am interested in this question because my current book project—a media archeology of the software bug—is seeking to raise awareness of and to question how much programming work in DH reproduces, intentionally or not, the power structures that are encoded by the programming practices we adopt from industry.

Slide This move is inspired by Mark Sample's intervention into the debate about DH and coding, where he claims that DH is meant to help spread the pleasures and perils of digital culture more evenly through the fields of humanisitic inquiry. However, Sample's model of DH necessitates thinking specifically about the practices of software-making we borrow from industry and the cultures of power we reproduce as a result.

Slide To ground this idea, I want to look at the famous distinction in computer science between composition and inheritance as models for software architecture and DH practice. Composition and inheritance are examples of what are called "design patterns" in software engineering, especially those branches of the field associated with object-oriented programming (OOP). Design patterns are repeatable organizations for OOP projects whose function are considered best practices (design patterns are also juxtaposed in the literature with dreaded "anti-patterns," or models of design that introduce inefficiency or ill functioning). In software engineering, composition is always to be preferred over inheritance (even though inheritance is often more widely taught in introductory computer science classes because it is more easily understood and more easily implemented). In this presentation, I will first define what both inheritance and composition are, before suggesting that much of DH work is focused on the former instead of the latter, and conclude by suggesting how and why we might better make composite design a part of our thinking about software for humanistic inquiry.

Inheritance

Slide Inheritance is a model implied by the nature of object-oriented software. In most OO languages, the type of a data object is defined by its membership in a class. Classes are categories of data, things like novels or animals or cars. Each specific data instance is instantiated from its class and inherits the class's interface (so, while books have a number of a pages, a particular book might have 245 pages; while animals might make a noise, a particular animal makes a honking sound).

Classes in OOP can also inherit characteristics from more abstract classes. That's what's going on in this chart here: Barnyard Animals offers a `makeNoise()` function while Barnyard Bird offers `flapWings()` and `layEggs()`. The specific classes of animals (Duck, Chicken, and Donkey) all override their inherited `makeNoise` function to produce their specific sound.

This pattern is inheritance. It's tied to the legacy of Aristotelean, hierarchical thinking in which everything can be classified and ordered; a model of the Universe in which everything makes sense and has a place.

Despite confirming our mental models of the universe, inheritance is extremely brittle. Adding or changing methods anywhere in the chain can produce all kinds of problems for other instances or less abstract classes. Moreover, what would happen if we suddenly didn't want to differentiate between birds and mammals, but instead wanted to inherit based on number of legs (two vs four)? Well, that whole diagram goes out the window!

Composition

Slide Which brings us to composition. In composition, what something is is less important than what it does.

If the diagram on this slide looks confusing, you're starting to get why composition is often ignored or deemphasized in teaching computer science. But let's try to untangle this.

Where the inheritance diagram was visually clear, it mapped a rigid hierarchy in which categories of things ("Barnyard Animals") were broken into more specific things

("Barnyard Birds" and "Barnyard Mammals") before being broken into the things themselves. In this diagram, we have two categories of objects: things and behaviors. The "Noise Making" behavior gives objects that possess it access to noise making (and each animal defines that behavior for itself), while Chickens and Ducks both get "Egg Laying" and "Wing Flapping." If a natural scientist wanted to come along after the fact and declare wing flapping and egg laying to be characteristics of "birds," so be it. We don't care; because we are thinking compositionally, we are only interested in defining behaviors and adding them to the various aspects of our program.

Composition, besides being the preferred and most widely recommended approach to problem solving in a number of computational domains, focuses on behavior instead of categorization and, therefore, focuses on building or reusing smart, powerful and, most importantly, focused tools for a particular job.

Composing DH

So why this discussion of computer science approaches to design? Thinking through this distinction transforms a key debate in DH on tool usage. In *Design Patterns*—the 1994 work that inaugurated the topic—, **Slide**inheritance is described as "white-box reuse" while **Slide** composition is described as "black-box reuse" (Gamma et al. 19). Remember that, here, composition, black-boxing, is strongly recommended. Such a recommendation is perhaps puzzling, given the tradition of denouncing black boxes in DH. **Slide** Joanna Drucker, for instance, has written of the "epistemological biases" of data visualization tools, in which charts and bar graphs claim to stand for reality but instead present intensely mediated visions of the real (Drucker 1). Inheriting a legacy from STS scholarship, part of DH practice has been invested in critiquing and opening black boxes.

So it would seem weird, at least it seemed weird to me, that this famous CS book is praising black boxes, which are, I think, such a dirty word in DH work. However, the authors of *Design Patterns* use “black box” a bit differently than we usually use it: in the white-box reuse of inheritance, “the internals of parent classes are often visible to subclasses”; while in the black-box reuse of composition, “no internal details of objects are visible.” Normally, not knowing how something works is seen as a bad thing in DH, so what gives?

Design Patterns uses “black box” to refer to how other objects in the software architecture relate, rather than the author of the code to the code, and this is an important distinction. In a white-box reuse situation, a piece of code explicitly depends on the interface it inherits from more abstract code. A black box reuse situation, however, produces a piece of software that does not depend on any other to be used. This means code is more easily reusable between projects.

Think, for instance, of why the selection of a content-management system is often so important for a public humanities project. Omeka gets you Neatline, but Drupal gets better user management (though it's kind of a nightmare to install). Wordpress is easy but not all that useful for doing much. Moreover, any plugins you use or write for any of those CMSs, are going to be lost if you have to switch. While this example probably isn't inheritance in the strictest sense as it's used in *Design Patterns*, the lesson is still worth thinking through: when you choose a tool ecosystem, you are committed to it's particular interface, you're bound to what you inherit.

SLOW DOWN

Moreover, a lot of DH practice is shaped by the kind of hierarchical thinking that drives inheritance. Library catalogs and genre theory are both moments in analog humanities that are driven by inheritance and the structure it provides. Similarly, a project like TEI strongly inscribes a tight, white-boxed data structure and demands users to think strongly about categorization in the way we were discussing earlier (is this a metaphor or a figure? what really *is* a poem?). We found a software architecture that ostensibly supported what we think we do, and we reproduced that model in digital space. However, is hierarchical thinking all we do in the humanities?

To answer my own question, I'm going to potential violate some DHSI decorum and talk about [**slide**] Jacques Derrida, specifically the idea of iterability he develops in "Signature, Event, Context." In that essay, Derrida challenges the idea of writing as conferring presence (specifically the presence of the author) by contrasting presence with difference (or *differance*, which both differs and defers differing). In this challenge, he constructs reading as iteration and writing as iterable. Each time we read, we read a different text, but each time we read, part of the text escapes and connects to the text as a whole, an aggregate effect.

In this play of differance Derrida highlights something important to how I think about a functional DH. **Slide** Richard Lanham, in "The Electronic Word," imagines a utopia of digitization that also foregrounds play as a textual act. Once texts are inside the computer, they become subject to iterations Derrida could not have imagined: remixed, mashed up, turned into bots, made into pictures, translated into gibberish by repeated trips through Google Translate.

When we get to thinking in terms of inheritance, we forget the personal nature of reading: our own individual iterations versus the stable sense of presence, context, and self-identity that Derrida is critiquing. Digital data, as Lanham reminds us, can be played with, in powerful ways. It can be taken apart and included with other functional components to make new and novel composites. I have been teaching students to play with our digitized literary archive in this fashion: to use texts and images and bits of software to produce new cultural forms and new ideas of criticality. Several students have made interactive fictions based on viral YouTube videos, while others generate fake country lyrics in proper ballad meter. I think this kind of experimental iteration, taking extant tools and literary archives as the raw material for new cultural forms, is an unexplored horizon for DH work, but it calls for a change in the way we think about what our software does, namely moving from inheritance to composition.

By taking small, well-functioning pieces that all work together with minimal coupling, the work of iteration with digital texts can continue beyond mere digitization. As more archives move online, we need to be thinking more about functionalizing these tools through APIs, open data, and interconnection. This kind of functionalization involves a fundamental acknowledgement that we may not even imagine how people might use our data, but we still need to think about how best to open that data, with a minimal of interface entanglements, so that we can better functionalize our projects and imagine a composite digital humanities.

Drucker, Johanna. “Humanities Approaches to Graphical Display.” *Digital Humanities Quarterly*, vol. 005, no. 1, Mar. 2011.

Gamma, Erich, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1 edition, Addison-Wesley, 1994.