Microsoft

MICROSOFT PROFESSIONAL

Object Thinking

David West

Microsoft

Object Thinking

David West

distributed. The traffic signal controls itself and notifies (by broadcasting as a different color) others of the fact that it has changed state. Other objects, vehicles, notice this event and take whatever action they deem appropriate according to their own needs and self-knowledge.

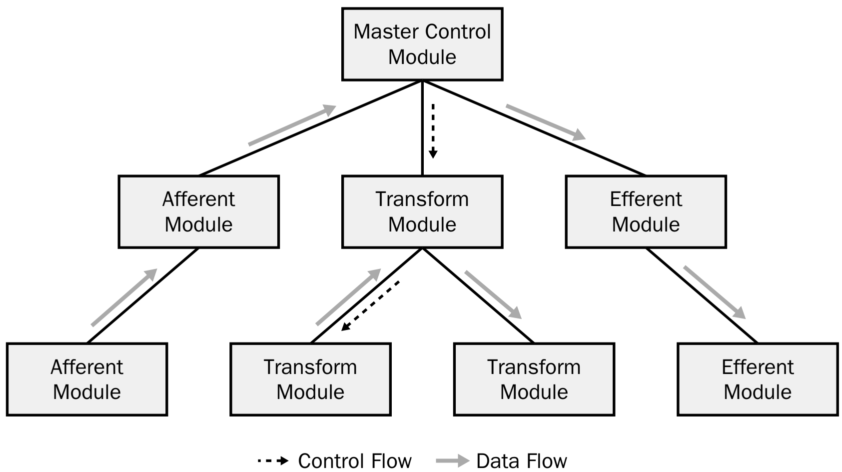

Figure 3-1 Program structure chart.

Note But what about intersections with turn arrows that appear only when needed? Who is in control then? No one. Sensors are waiting to detect the "I am here" event from vehicles. The traffic signal is waiting for the sensor to detect that event and send it a message: "Please add turn arrow state." It adds the state to its collection of states and proceeds as before. The sensor sent the message to the traffic signal only because the traffic signal had previously asked it to—registered to be notified of the "vehicle present" event. Traffic management is a purely emergent phenomenon arising from the independent and autonomous actions of a collectivity of simple objects—no controller needed. If you have a large collection of traffic signals and you want them to act in a coordinated fashion, will you need to introduce controllers? No. You might need to create additional objects capable of obtaining information that individual traffic signals can use to modify themselves (analogous to the sensor used to detect vehicles in a turn lane). You might want to use collection objects so that you can conveniently communicate with a group of signals. You might need to make a signal aware of its neighbors, expanding the individual capabilities of a traffic signal object. You will never need to introduce a "controller."

Eliminating centralized control is one of the hardest lessons to be learned by object developers.

Object Principles—Software Principles

Stating and explaining object presuppositions is important. It is also important to show the relationship between those principles and generally accepted principles of software design criteria. Exploring that relationship will further explain and illustrate the object principles and show how they recast thinking about design without rejecting traditional design goals.

Witt, Baker, and Merritt have written an excellent encapsulation of the fundamental ideas about software design and architecture.[8] Chapter 2 of their book identifies a set of generally accepted axioms and principles that define software quality:

- **Axiom of separation of concerns** Solve complex problems by solving a series of intermediate, simpler problems.
- **Axiom of comprehension** Accommodate human cognitive limitations.
- **Axiom of translation** Correctness is unaffected by movement between equivalent contexts.
- **Axiom of transformation** Correctness is unaffected by replacement with equivalent components.
- **Principle of modular design** Elaborates the axiom of separation of concerns.
- **Principle of portable designs** Elaborates the axiom of translation.
- **Principle of malleable designs** Provides the means for compositional flexibility.
- **Principle of intellectual control** Appropriate use of abstractions.
- **Principle of conceptual integrity** Suggests a limited set of conceptual forms.

Few would argue with these axioms and principles, although they would certainly argue about the appropriate means for realizing them. Object thinkers strive to achieve the goals implied by these axioms and principles as much as

8. Witt, Bernard I., F. Terry Baker, and Everett W. Merrit. *Software Architecture and Design: Principles, Models, and Methods*. Van Nostrand Reinhold, 1994.

any other software developer *and* believe that objects provide the conceptual vehicle most likely to succeed.

For example, the separation-of-concerns axiom and the principle of modularity mandate the decomposition of large problems into smaller ones, each of which can be solved by a specialist. An object is a paradigmatic specialist. Large problems (requiring a number of objects, working in concert to resolve the problem) are decomposed into smaller problems that a smaller community of objects can solve, and those into problems that an individual object can deal with. At the same time, each object addresses the principles of intellectual control (individual objects are simple and easy to understand) and the principle of conceptual integrity (there should be a small number of classes). Properly conceived, an object is a natural unit of composition as well. An object should reflect natural, preexisting decomposition ("along natural joints") of a large-scale domain into units already familiar to experts in that domain. Conceived in this fashion, an object clearly satisfies the principle of intellectual control. Objects will also satisfy the principle of conceptual integrity because there will be a limited number of classes of objects from which everything in the domain (the world) will be constructed. In Chapter 4, "Metaphor: Bridge to the Unfamiliar," an argument will be presented suggesting that the total number of objects required to build anything is around 1000.

Objects are designed so that their internal structure and implementation means are hidden—encapsulated—in order to satisfy the axiom of transformation and the principle of portable designs.

The principle of malleable designs has been the hardest one for software to realize: only a small portion of existing software is flexible and adaptable enough to satisfy this principle. In the context of object thinking, malleability is a key motivating factor. Object thinkers value designs that yield flexibility, composability, and accurate reflection of the domain, not machine efficiency; not even reusability, although reusability is little more than cross-context malleability.

In fact, it might be said that for object thinkers, all the other axioms and principles provide the means for achieving malleability and that malleability is the means whereby the highest-quality software, reflective of real needs in the problem domain, can be developed and adapted as rapidly as required by changes in the domain. Agile developers and lean developers[9] value malleability as highly as object thinkers. XP software systems emerge from software that satisfies the demands of a single story, an impossibility unless it is easy to refactor, adapt, and evolve each piece of software (each object); impossible unless each bit of software is malleable.

---

9. Poppendieck, Mary, and Tom Poppendieck. *Lean Software Development: An Agile Toolkit for Software Development Managers*. Addison-Wesley. 2003.

Fred Brooks wrote one of the most famous papers in software development, "No Silver Bullet: Essence and Accidents of Software Engineering."[10] In that paper, he identified a number of things that made software development difficult and separated them into two categories, *accidental* and *essential*.

Accidental difficulties arise from inadequacies in our tools and methods and are solvable by improvements in those areas. Essential difficulties are intrinsic to the nature of software and are not amenable to any easy solution. The title of Brooks's paper refers to the "silver bullet" required to slay a werewolf—making the metaphorical assertion that software is like a werewolf, difficult to deal with. Software, unlike a werewolf, cannot be "killed" (solved) by the equivalent of a silver bullet.

Brooks suggests four essential difficulties:

- **Complexity** Software is more complex, consisting of more unlike parts connected in myriads of ways, than any other system designed or engineered by human beings.
- **Conformity** Software must conform to the world rather than the other way around.
- **Changeability** A corollary of conformity: when the world changes, the software must change as well, and the world changes frequently.
- **Invisibility** We have no visualization of software, especially executing programs, that we can use as a guide for our thinking.

He also investigates potential silver bullets (high-level languages, time sharing, AI, and so on) and finds all of them wanting. Object-oriented programming is considered a silver bullet and dismissed as addressing accidental problems only.

Although I would agree with Brooks in saying that *object technology*—languages, methods, class hierarchies, and so on—addresses only accidental problems, *object thinking* does address essential difficulties, and it does so with some promise. Objects can conform to the world because their design is predicated on that world. Objects are malleable, resolving the changeability issue. Objects provide a way to deal with the complexity issue and even allow for the emergence of solutions to complex problems not amenable to formal analysis. The metaphors presented in Chapter 4 provide the tools for visualization to guide our thinking.

Object thinking suggests we deal with software complexity in a manner analogous to the ways humans already deal with real-world complexity—using behavior-based classification and modularization. Object thinking is focused on the best means for dealing with conformity and changeability issues—the

10. IEEE Computer, April 1987.

malleability principle—as a kind of prime directive. And invisibility is addressed, not with an abstract geometry as suggested by Brooks, but via simulation (working software using an XP perspective)—direct, albeit metaphorical, simulation of the real world. If we can understand the complex interactions of objects in the real world (and we do so every day), we should be able to visualize our software as an analogous interaction of objects.

Forward Thinking

Communication and Rules

Because the UVM might be dispensing food items and because we want the customer experience to be always positive, we want to ensure that no spoiled products are vended. This leads to a story—*Expire: no product is sold after its expiration date has been reached*.

The development team discusses (and codes) various ways this might be accomplished. Through a combination of refactoring efforts and arguments, it is decided that the expiration problem will best be solved by a group of objects communicating with one another, with those communications being triggered by events.

Whenever a product is placed in the vending machine, it asks itself for its expiration date. It then asks the *SystemClockCalendar* to add an *eventRegistration* (consisting of the *productID* and the "die" message) for the event generated whenever a new day is recognized by the *SystemClockCalendar*. (Programmers, even extreme programmers, often have a rather grim sense of humor; hence the "die" message to effect product expiration.) At the same time, the *Dispenser* object asks the new product to accept a registration for the "I'm dead" event that the product will generate when it receives the "die" message from its own event registration that was placed with the *SystemClockCalendar*. The dispenser's *eventRegistration* with the product will cause the message "disableYourself" to be sent to the dispenser, who will, indeed, "disable" itself (with the accompanying event that other objects—such as the menu or the dispenser collection—might register for).

Breaking up a potentially complex decision-making and cascading-effects problem into pieces that can be distributed among many objects while at the same time relying on simple, reusable components such as an *eventRegistration* and a *Dispatcher* greatly reduces the complexity that worries Brooks. It also accommodates the conformity and changeability requirements imposed on software: event registrations can be added or

Forward Thinking *(continued)*

deleted as needed, redirected to other objects, or transformed so that the registering object receives different messages at different times without the need to rewrite and recompile source code.

The development team found another opportunity for simplification as they worked with various types of rules that governed actions in different parts of the UVM. One kind of rule was, "Don't vend a product unless sufficient funds have been accumulated." Another rule was, "Refund money using the largest coins available, moving to lower-denomination coins only when the larger denomination is greater than the sum yet to be refunded."

Using a combination of refactoring and *appropriate* abstraction, the development team defined and designed a rule object. (XP philosophy warns against premature abstraction: abstraction that is not derived from refactoring, meaning not grounded in efforts to achieve simplification. Hence the adjective *appropriate* in the preceding sentence.) A rule is an ordered collection of constants, variables, and operations. A variable consists of an object and a message to be sent to that object. When a variable sends the message to the target, the resultant value replaces the unknown value of the variable. When asked to evaluate to a result, a rule iterates across its elements, asking each variable to instantiate itself to a real value, and then applies the operators to the instantiated variables and constants.

All four of Brooks's concerns about software's essential difficulties are addressed: simplification of complexity, ease of conformity and adaptability, and visualization. A rule is an easy thing to visualize—we see examples of them in everyday life frequently—and the process of instantiation and resolution is very straightforward—we can see it operating in our mind's eye with no difficulty. Reliance on simulation constantly provides other visualizations of the software we are creating.

Cooperating Cultures

Arguing for the existence of an object paradigm or object culture is not and should not be taken as an absolute rejection of traditional computer science and software engineering[11]. It would be foolhardy to suggest that nothing of value has resulted from the last fifty years of theory and practice.

11. The ideas in this section were first published in a short editorial by the author in *Communications of the ACM*, 1997.