Research

On the Succinctness of Idioms for Concurrent Programming

David Harel[1], Guy Katz[1], Robby Lampert[2], Assaf Marron[1], and Gera Weiss[3]

1 Weizmann Institute of Science, Rehovot, Israel
2 Mobileye Vision Technologies Ltd., Jerusalem, Israel
3 Ben Gurion University, Beer-Sheva, Israel

2 Definitions

2.1 The Request-Wait-Block Model

In this work we focus on the *Request-Wait-Block* ($\mathcal{RWB}$) model for concurrent programs. As we mentioned before, the requesting, waiting-for and blocking idioms are common and appear in various models such as *publish-subscribe* architectures [6], *live sequence charts* [4] and *behavioral programming* [13]. Further, research has shown that these idioms often enable programmers to specify and develop systems naturally and incrementally, with components that are aligned with how humans often describe behavior [7, 13]. Still, the $\mathcal{RWB}$ model is not intended to be programmed in directly – rather, it is intended as a formal representation of programs written in higher level languages, for the sake of rigorous analysis.

The formal definitions of the $\mathcal{RWB}$ model are as follows. An $\mathcal{RWB}$-*automaton* consists of orthogonal components called $\mathcal{RWB}$-threads:

▶ **Definition 1.** A *Request-Wait-Block-thread* ($\mathcal{RWB}$-*thread*) is a tuple $\langle Q, \Sigma, \delta, q_0, R, B \rangle$, where Q is a finite set of states, Σ is a finite set of events, $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation and q_0 is an initial state. We require that δ be deterministic, i.e. $\langle q, e, q_1 \rangle \in \delta \wedge \langle q, e, q_2 \rangle \in \delta \implies q_1 = q_2$. For simplicity of notation, we use $\bar{\delta}$ to indicate the effect event e has in state q (or its absence):

$$\bar{\delta}(q, e) = \begin{cases} q' & ;\text{if exists } q' \in Q \text{ such that } \langle q, e, q' \rangle \in \delta \\ q & ;\text{otherwise}. \end{cases}$$

Scenario-Based Programming: Reducing the Cognitive Load, Fostering Abstract Thinking *

Giora Alexandron Michal Armoni Michal Gordon David Harel
Weizmann Institute of Science, Rehovot, 76100, Israel

ABSTRACT

We examine how students work in scenario-based and object-oriented programming (OOP) languages, and qualitatively analyze the use of abstraction through the prism of the differences between the paradigms. The findings indicate that when working in a scenario-based language, programmers think on a higher level of abstraction than when working with OOP languages. This is explained by other findings, which suggest how the declarative, incremental nature of scenario-based programming facilitates separation of concerns, and how it supports a kind of programming that allows programmers to work with a less detailed mental model of the system they develop. The findings shed light on how declarative approaches can reduce the cognitive load involved in programming, and how scenario-based programming might solve some of the difficulties involved in the use of declarative languages. This is applicable to the design of learning materials, and to the design of programming languages and tools.

described abstraction as "the only mental tool by means of which a very finite piece of reasoning can cover a myriad of cases" (p. 864). Wing [39] referred to abstraction as one of the defining characteristics of computational thinking, and as a core skill that a computer scientist must possess. According to the task force chaired by Denning [8], abstraction is one of the three processes that characterize the discipline.

Hazzan and Kramer [25] defined the concept of abstraction in computer science (CS) and software engineering (SE) as "a cognitive means, according to which, in order to overcome complexity at a specific stage of a problem solving situation, we concentrate on the essential features of our subject of thought, and ignore irrelevant details." (p. 3). Abstraction is fundamental to many CS and SE subjects, but specifically, it is central to programming. Thus, an essential characteristic of programming languages is their approach to abstraction.

In [11], Green proposed a cognitive framework for characterizing programming languages, termed 'cognitive dimensions', which also refers to abstraction properties. We find

Programming with the User in Mind *

Giora Alexandron[1] Michal Armoni[1] David Harel[1]

Weizmann Institute of Science, Rehovot, 76100, Israel
{giora.alexandorn, michal.armoni, david.harel}@weizmann.ac.il

Keywords: POP-II.B.scenario-based design, POP-III.C.visual languages, POP-III.D. specification languages, POP-V.A.problem solving

Abstract. In this paper we present preliminary findings regarding the possible connection between the programming language and the paradigm behind it, and programmers' tendency to adopt an external or internal perspective of the system they develop. According to the findings, when working with the visual, inter-object language of *live sequence charts (LSC)*, programmers tend to adopt

"behavioral programming harel"

http://www.wisdom.weizmann.ac.il/~harel/papers.html

⚠ Cons

Y **Hacker News** new | threads | past | comments | ask | show | jobs | submit sktrdie (2416) | logout

* Depending Less on Structure (lmatteis.github.io)
62 points by sktrdie 21 days ago | hide | past | web | favorite | 51 comments

▲ hacker_9 21 days ago [-]

1. Example is too simple, functions could modify anything in a model depending on context.

2. Good luck trying to optimise this (just merging all behaviours together at the end does not equal optimisation).

3. You're just pushing the problem elsewhere - now I need to understand all behaviours from a black box, then wrap them, all while dealing with all the other wrappers people have already put in place. Imagine dealing with that complexity.

Y **Hacker News** new | threads | past | comments | ask | show | jobs | submit

* B-threads: programming in a way that allows for easier changes (medium.com)
167 points by sktrdie 6 months ago | hide | past | web | favorite | 92 comments

> However I think that this approach comes with some cost. For instance if (isMultipleOfThree(x) && endsWithDigitFive(x)) is very easy to understand, but isMultipleOfThree(x) and endsWithDigitFive(x) in different modules is a lot harder. Personally I see this more as a trade-off: trading control flow for extensibility.

I think this is a massive understatement. Imagine replacing most of the ifs in your program with new top-level constructs that run in parallel and exchange events. The maintenance overhead of this idea seems astronomical, for relatively little benefit. I would be terrified to add new modules to such a beast, knowing that they could affect the behavior of any other module, in complicated cascades.

https://news.ycombinator.com/item?id=22033987
https://news.ycombinator.com/item?id=20556217

I'm going to make a conscious effort not to come off sounding like an asshole, but please excuse me if I slip up. I find many of the ideas in post to be fundamentally at odds with the direction that "good software development" should be travelling. The core of my feeling is best captured by the following quote from the post:

>As a system grows in complexity we don't necessarily care about how old b-threads have been written, hence we don't care about maintaining them.

This post is essentially formalizing the process of creating a Big Ball of Mud[0] that is so complex and convoluted that it is impossible to understand. The motivation for formalizing this process seems sane and with good intentions: to add functionality quickly to code you don't really understand. Normally, doing something like is considered cutting corners and incurring explicit technical debt, and must be used sparingly and responsibly. However, the process of "append-only development" is embracing the corner cutting and technical debt as a legitimate development process. I can't get on board with this.

To be more specific, with an example (and maybe I am wrong in understanding the post, this would be the time to point that out to me), let's suppose you have a massive complex software system that was built over the years with this "append-only" style of development. One day you find a nasty bug in one of the lower layers, and to correct it, you have to change some functionality, which moves/removes some events that subsequent layers are depending on for their own functionality. Suddenly, you are faced with rewriting all of those layers in order to adapt to your bugfix. What you're left with is a nightmare of changes that disturb many layers of functionality, because they're all based on this append-only diffing concept: the next layer is dependent on the functionality of the previous layer.

This is what programming APIs are for: to change functionality in lower layers with minimal influence subsequent layers. This post and process seems to be imagining a world without APIs.

0. http://www.laputan.org/mud/

Nothing about this approach seems easier to reason about nor maintain over the long term. Transactional integrity, migrating data structures, and modifying business logic/control flow are key building blocks required for many production applications, certainly line of business ones.

I'm reminded of the golden hammer fascination with event sourcing, and I continue to see event sourcing applied inappropriately. Is it useful for some specialized applications? Yes, and it's a great tool there! But should it be the default tool you reach for to solve most problems over a relational DB and your most trusted, high level, stable programming language with a mature library ecosystem?

No to that question, and no to this.