**Richard Steiger**

Content

[Ensemble Innovations i](#_Toc132816680)

[1 Overview 1](#_Toc132816681)

[1.1 Purpose 1](#_Toc132816682)

[1.2 Open Issues 1](#_Toc132816683)

[Part 1: Theory 2](#_Toc132816684)

[2 Good Abstraction Yields High Design Consistency 2](#_Toc132816685)

[3 Design Granularity Philosophy 2](#_Toc132816686)

[4 A New Frame Around Information and Representation 2](#_Toc132816687)

[5 Type-Centric Architecture 2](#_Toc132816688)

[6 Polylinguistic Synergy 3](#_Toc132816689)

[7 Advanced Actor Foundation 3](#_Toc132816690)

[8 Component Societies 3](#_Toc132816691)

[9 Flows 3](#_Toc132816692)

[10 Lifecycles 4](#_Toc132816693)

[11 Addressing 4](#_Toc132816694)

[12 Ditching Builds 4](#_Toc132816695)

[13 The Universal Canvas 4](#_Toc132816696)

[13.1 A Diagram Is Worth A Lot 4](#_Toc132816697)

[14 Z: The Last Letter In Languages 5](#_Toc132816698)

[15 MetaProgramming 5](#_Toc132816699)

[16 Functors, Monads, and Gonads 6](#_Toc132816700)

[17 Distribution 6](#_Toc132816701)

[17.1 Replication 6](#_Toc132816702)

[17.2 Ubiquity and Worlds 6](#_Toc132816703)

[18 Modularity and System Blueprints 6](#_Toc132816704)

[19 Skating Over Software and System Lifecycles 6](#_Toc132816705)

[Part 2: Operations 7](#_Toc132816706)

[20 Modules, Components, Glue 7](#_Toc132816707)

[21 AI/ML Playgrounds 7](#_Toc132816708)

[22 Essential Process Model Dimensions 7](#_Toc132816709)

# Overview

## Purpose

This document is a marketplace for Ensemble’s innovative ideas.

It’s purpose is to provide an initial forum for

* + announcing and revealing novel concepts discovered during the last couple of decades Eworks development
  + to organize such concepts to create an Innovation Ontology
  + to identify user groups who are potential beneficiaries of each conceptual center, and to sketch the features, benefits, and advantages of each
  + to serve as a workbench and agora for analyzing, comparing, quantifying benefits, and otherwise facilitating identification of the top few initial opportunities on which to concentrate ESS’s near-term execution and delivery.

## Open Issues

Part 1: Theory

This section is intended to capture the key ideas, insights, and design decisions underlying the entire Eworks codebase.

# Good Abstraction Yields High Design Consistency

* well-abstracted algorithms, patterns, modules, libraries, and other executable artifacts inherently yield minimally-redundant, hence highly-composable parts
* in design, abstraction’s gravitational field migrates functions to the optimum types, and minimizes overrides, yielding a maximally readable, easily understandable, and rapidly assemblable solutions
* it is a challenge to overstate the power that clean, minimal abstractions, or more precisely, systems comprising such abstractions, and strongly guided by principles aimed at optimizing such systems’ readability, modularity, reusability, simplicity, comprehensability, performance, adaptability, productivity-boosting meta-tooling, etc.

# Design Granularity Philosophy

* given the implementation choice between
  + creating many little classes that finely-divide responsibilities, versus
  + creating bulletproof, hardened, complete, deeply-specified and understood types
* a core eworks design principle encourages picking the coarser-grain but heftier second pattern

# A New Frame Around Information and Representation

* no bit of “information” (data) has any meaning short of a representation
* Eworks treats every unit of information as a [data, representation] pair
* Eworks is unique in treating representation on an equal footing as data being represented
  + it provides a rich vocabulary of representations, based on an underlying algebra of component representations, plus various kinds of compositions, allowing arbitrarily rich and complex presentations to be easily be constructed, from a compact set of underlying primitives
  + it provides interactive facilities for fluently modifying existing representation, and creating alternate new representations, on-par with parallel data modifications
  + the typical use-case for this overall capability is for users to co-evolve queries, filters, projections, and other data-level computations, and their associated representations, in a fluid, facile, rapidly-improving, and overall effective “dance with data” that rapidly reveals patterns, trends, and hidden insights in the data

# Type-Centric Architecture

* Eworks is profoundly, deeply, sometimes even surprisingly *type-centric* (aka *type-driven*):
  + this is the inevitable implication of the above dominant requirements for radical, maximal abstraction
  + put differently, millions of human-hours have been expended on determining what are the top few most-productive platform architectures, specifically the constraints, force vectors, and other factors are all orthogonal
  + the fact is that most information processing functionality is easily modeled (represented) by finite constructions of well-known ***basic functional modules***

# Polylinguistic Synergy

* by providing a common core program and semantic representation that spans leading languages, including Java, JavaScript, Python, C, C++, C#, Kotlin, and Scala, to create a *code agora* in which programs, libraries, scripts, and fragments may be relatively seamlessly integrated to rapidly assemble value-generating solutions
* by enabling low-friction interoperation of software assets, created in diverse languages, but treated as a combined mega-library of computational and application-spanning assets, the solution development and delivery velocity can be expected to be at least an order of magnitude faster than the current mono-linguistic straight-jacket imposed by today’s development platforms
* the key underlying technologies enabling this polylinguisic synergy include
  + unified type system
  + universal ASTs
  + language-agnostic program editing and visualizing tools

# Advanced Actor Foundation

* Eworks’ core implementation, and all application-level types, are anchored in an advanced and disruptively powerful generalization of the original Actor formalism developed by Hewitt, Steiger, and other members of the Actor Group at MIT’s AI lab in the 70s

# Component Societies

* a key architectural principle is to build the system out of components, each focussed on solving, implementing, managing, or otherwise address a well-defined scope of conern, and to freely leverage its peer components to address it’s remaining needs
* in this view, each component serves in a service provider (data producer, specification authority, …) role, and equally act in the role of a consumer of other services

# Flows

* Eworks introduces the Flow concept
  + replaces iterators and streams, having super-signature
    1. simpler, more powerful, cleaner, more easily extended
  + radically composable
  + universal: all collections, processes, I/Os, etc have well-engineered, tuned implementations
  + flows are among the few maga-leveraging concepts, anywhere:
    1. flows are radically composable
    2. a rich set of operators are defined on flows (among other types)
    3. many typical queries and processes can be captured in 1-2 lines of flow code
* each flow is potentially also a cursor
  + cursors add bidirectional movements, and random-access
  + by being defined at the core tier, there’s tremendous flexibility to propagate funSpecs downward
* flow funSpecs provide powerful value up and downn the runtime pipeline
  + by being able to listen for and leverage functional changes, lower-level tiers are often able to instantiate far more performant implementations than is possible without being meta-informed

# Lifecycles

* the technical function of lifecycle management ideally includes encapsulating creation, initialization, graph template expansion, composite assembly, dependency injection, resource management, access control, backup/mirroring, and security
* the user’s objective is for lifecycle management to be as automatic and invisible as possible
* the developer’s objective is similar, for the platform to implicitly handle all lifecycle concerns
* by meeting the above objectives, components are freely composable, without requiring a single line of executable code
* in particular, this in turn powers drawing-based flow/pipeline design studios

# Addressing

* every actor has an address

# Ditching Builds

* in contrast to nearly all other system-building technology platforms, any underlying program transformations, binary representations, and associated build processes are rendered all but totally invisible
* the perspective here is that such processes serve to merely transforms programs from source form to more efficiently executable forms, something that can be completely automated
* since such lower-level representations provide no essential knowledge nor other semantic value, having them surface in the development and system management experience is only permitting noise, thereby subtracting value
* the eworks platform treats all code transforms as continuously executing mappings, represented as flows
* when any part of the system’s blueprint (code, configuration, policies, constraints, etc) is modified, a sophisticated transform flow is triggered that, in the predominant happy case, seamlessly alters runtimes to conform

# The Universal Canvas

## A Diagram Is Worth A Lot

* developers, devops, SREs, product managers, and every other human interacting with systems thinks in a dynamic mixture of discrete, visualized networks of structures
* eworks bridges the pervasive mind-machine gap by making diagrams a first-class program and system design representation

# Z: The Last Letter In Languages

* As numerous illuminaries have commented over the last 15 years, Java is both outstandingly powerful, and has increasingly fallen behind programming language innovation in the broader community
* Java’s core architecture occupies a sweet-spot in runtimes, providing a language-agnostic JVM (Java virtual machine), and driving, flagship language
* by Oracle’s java team’s own discussions, while Java has significantly evolved over the last several years, it’s laudible and strong commitment to preserver backwards-compatability is both one of its strengths, but inevitably also is progressive weakness to incorporate crucial, productivity-scaling lessons and innovations on other languages
* an illustative set of Z’s innovations and improvements includes
  + complete runtime type systems for Java, Python, others?
    1. basis for all metaprogramming
    2. most tools are type-driven
    3. tools are built from templates
  + public hooks into AST generation and processing are provided
    1. complete freedom to define assignment semantics, within given scope
  + smart casts, Elvis operator
  + far more complete/thorough type inferencing
  + type aliases, abstract types
  + self and dynamic types
  + templating
    1. grok “That nice little backtick for templating”
  + middleware functionality via annotatation
    1. transactions, persistence, constraints, verification, evolution, event generation, event reception,
    2. this mechanism is key to radically reducing (lines of) code, since is declarative
    3. when annotations are parametric, a dependency graph is induced
    4. relying solely on textual specifications/descriptions/annotations challenges readers’ modeling capacity
    5. a tier-zero eworks tool capability is graphical editors, intended to make deep understanding and agile editing possible
  + structural types
  + Native JSON type for objects
* Z ReferenceEditor breaks new ground in viewing, browsing, producing, and perfecting programs, viewed as artifacts that comprise code, data, and information (represented)

# MetaProgramming

# Functors, Monads, and Gonads

* Eworks introduces the Flow concept
  + much like, but entirely different from, Java8 streams
    1. simpler, more powerful, cleaner, more easily extended
  + replaces iterators and streams
* each flow is potentially also a cursor
  + cursors add bidirectional movements, and random-access
  + by being defined at the core tier, there’s tremendous flexibility to propagate funSpecs downward
  + by being able to listen for and leverage functional changes, lower-level tiers are often able to instantiate far more performant implementations than is possible without being meta-informed

# Distribution

## Replication

* unless othersize limited, actors may proliferate replicas across sites, typically by being passed as an argument to a method-call, or returned as the result of such a c all
* the resulting simplicity and freedom streamlines distributed programming to largely overlap local, non-distributed programs
* there’s an extensive library of containers, collections, and higher-order function templates/generators, that recent end-app developments have shown cover a wide swath of typical app component requirements

## Ubiquity and Worlds

* a distinguished, univeral set of actors are ubiquitous, i.e. unique, and present in every site
* ubiquitous actors are the “globals” in eworks
* a world is a collection of sites that are mutually familiar, easily addressed, readily available
* the \_World type has core operations for connecting, ingesting, splitting, and seperating worlds

# Modularity and System Blueprints

# Skating Over Software and System Lifecycles

Part 2: Operations

# Modules, Components, Glue

# AI/ML Playgrounds

* Z has proven to be a vastly-superior platform for building AI and related apps
* Z has been and increasingly is able to leverage the ML Python communities capabilities
  + seamless, either-direction Z-Python calls and data replications
  + over time, a growing library of transpilations of key Python

# Essential Process Model Dimensions

* to be usable and effective, models of any process must be based on the following underlying dimensions:
  + time
  + artifact structure
  + priority
  + conceptual focus
* the Brainamp product …