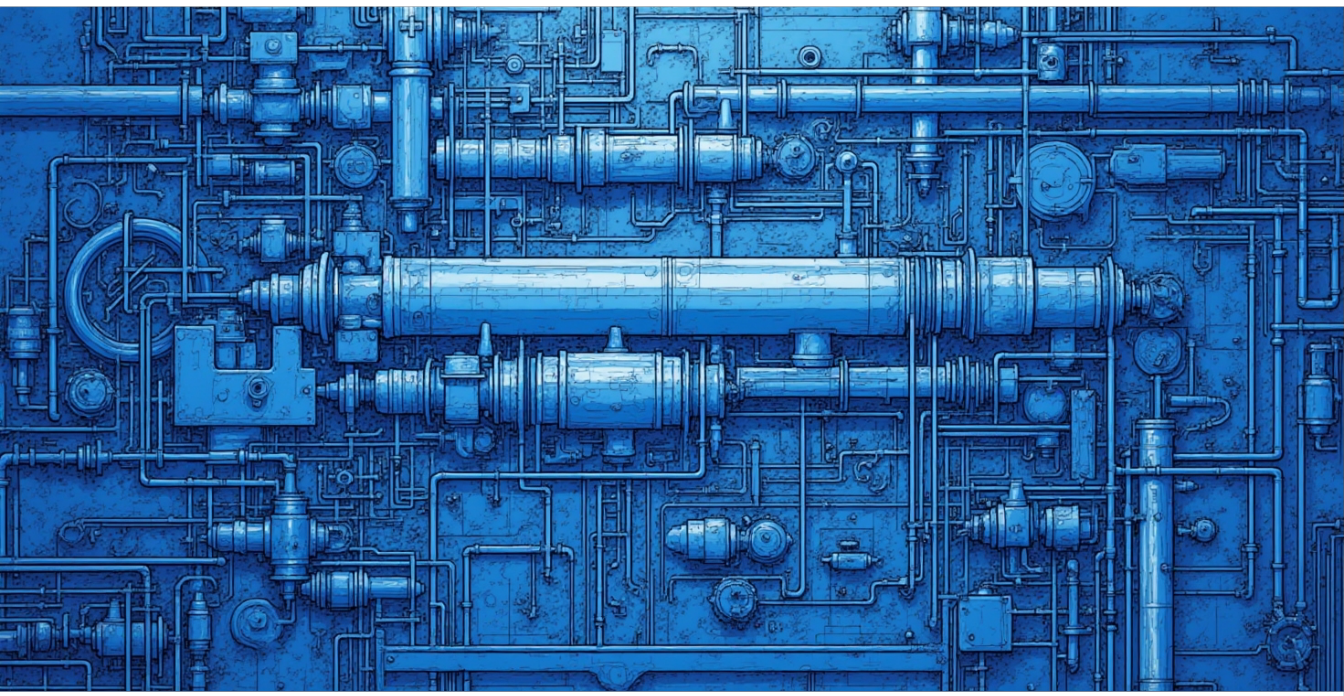
General-Purpose Visual Programming Language

Pipe

Pragmatic Approach to Visual Programming



Feature-rich visual language with highly detailed design all the way down to API level

Designed to supplement and enhance AI code generation

Maximizing productivity of AI-driven software development via visual programming

Oleg P. Kabanov

General-Purpose Visual Programming Language

Pipe

Pragmatic Approach to Visual Programming

Oleg P. Kabanov

For licenses and attributions of all fonts used in this book, please see the end of the book.

ISBN: 978-1-0696277-0-4

First Edition

Published by Oleg Pavlovitch Kabanov

Please feel free to leave your comments or review at http://www.pipelang.com

Contents

This page is intentionally left blank

1. Introduction

1.1. Introduction

Visual programming makes it easier for people of different backgrounds and levels of expertise to create software. Visual languages differ from text-based (non-visual) languages as they are based on a graphical notation. However, text-based languages such as C# and Java are the most used and popular today. Despite many efforts to create a general-purpose visual language, there isn't any practical and widely used one. Text-based programming languages still dominate as a primary method of software production nowadays.

To address the need in visual programming tools, low-code platforms are gaining popularity these days. They provide a visual method of software construction. However, each of these platforms has its own non-generic graphical notation, which means the problem of defining a common visual language for all low-code platforms is a significant challenge for the software development industry.

AI code generation technologies pushed software development productivity to the next level, and it seems visual languages became irrelevant. However, visual programming can still play a significant role when combined with AI code generation. Even if AI is capable to comprehend complex technical and business specifications, there will always be ambiguities and gaps in prepared specifications for code generation, and number of potential incorrect assumptions made by AI during code generation are going to be increasing almost exponentially as complexity of a project grows. Providing detailed enough specifications for AI is a human responsibility and it is an extremely difficult task. Another words, complete and unambiguous explanation of requirements to AI becomes more and more difficult as complexity of a project increases. The solution of this problem is generating code only for base-level components easily explainable to AI, completing the rest of application via manual programming. However, it defeats the purpose of using AI to eliminate human coding. This is where visual programming can be extremely useful as an alternative to text-based languages, boosting AI-driven software development productivity much further.

The next stage of AI-assisted visual programming is probably going to be a direct generation of visual flowcharts. This will significantly simplify human job of verification and understanding generated logic, also making it very easy to change generated visual workflow manually as a faster and easier alternative to resorting to full code re-generation every time modifications of logic are needed.

All these circumstances point to a need for a practical general-purpose visual programming language. Therefore, this book introduces a new visual language "Pipe". The book consists of two main parts: Chapter 2 "Language Specification" and Chapter 3

"Design Rationale and Future Development". Chapter 2 is a formal specification of the Pipe language. Chapter 3 provides summary of a Pipe language specification, also explaining underlying reasons of the language design decisions and describing new ideas for future development of the language. Example of Pipe diagram for a real business case is provided in the Figure 65.

All non-visual programming language examples are using C++. Definitions of new terms in Chapter 2 are highlighted by **underline bold**. Names of elements from diagrams, figures, tables, and source code fragments are shown by ***bold italic***. If a word or sentence needs to be highlighted for any other reason, then it is shown in **bold**. C++ source code and domain/object specifications (see 2.1.4.1.2 "Domain Specification" and 2.1.4.1.3 "Object Specification") use a `fixed-width` font.

Please use the following link to leave feedback related to this book or about visual language Pipe in general: http://www.pipelang.com.

1.2. Language Overview

Pipe is a visual programming language where citizen developers create workflows from a set of visual building blocks by combining and connecting them together. Pipe has the following features:

- ✓ **Open visual language.** Developers are not constrained by a fixed set of prebuilt components with limited customizations, and they can create or modify visual components exactly to their requirements and specifications.
- ✓ **General-purpose visual language.** The language contains only general-purpose abstract elements. There are no elements representing narrow domain-specific concepts or notions, making Pipe a truly general-purpose visual language.
- ✓ **Compact but powerful language.** Pipe provides relatively few elements and concepts, but this small element base contains expressive and versatile visual building blocks for implementing a wide variety of algorithm.
- ✓ **Practical visual language.** Pipe does not replace non-visual programming languages but rather complements them, leaving lower-level component development to traditional languages and providing visual methods to combine programmed components into workflows.
- ✓ **API for integration with non-visual languages.** Complete API specification for integration with non-visual languages is an essential part of Pipe language specification. This API can be consumed by any object-oriented non-visual programming language used for integration with Pipe workflows.
- ✓ **Comprehensive and detailed language specification.** Comprehensive and detailed language specification makes building a complete virtual machine for Pipe flowchart execution a straightforward task, as precise and unambiguous description is provided for all language elements.

- ✓ **Statically typed visual language.** Pipe is a statically typed visual language similar to top-tier non-visual programming languages such as Java, C#, C/C++, etc.
- ✓ **Levels of usage.** It is not required to know the complete Pipe specification for software development as multiple levels of usage make it possible to start building Pipe workflows with just a partial skill set.
- ✓ **Integration with AI code generation tools.** Source code produced by AI code generation tools can be placed inside visual components for Pipe integration. Therefore, visual workflow builder can play a role of a composition and integration layer for AI-generated code converted into reusable visual components.
- ✓ **Next generation of low-code platforms.** Pipe usage as integration layer of visual components encapsulating AI-generated code can inspire the next generation of low-code platforms where users are not constrained by prebuilt components with limited customizations anymore as they can generate new components using AI, integrating them via visual language Pipe.
- ✓ **Long-term vision.** While the current version of Pipe language already provides a great number of features, there are lots of new ideas and features planned for future versions of the language.

2. Language Specification

2.1. Concepts

2.1.1. Pipeline

Pipeline is a directed graph connecting pipeline **inputs** with **outputs** directly or via intermediary **components** represented by graph nodes (see Figure 1). Each component may have any number of its own inputs and outputs called collectively **pins**.

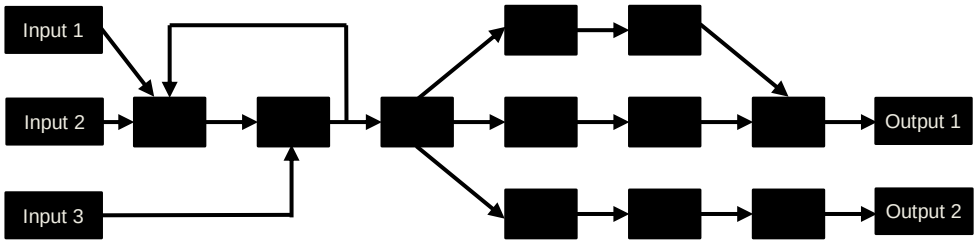


Figure 1. Pipeline as a directed graph.

2.1.2. Components

2.1.2.1. Component Taxonomy

The following top-level components are defined in Pipe language (see Figure 2):

- **Runlet** – data processing unit implementing some functionality.
- **Memlet** – data storage component.

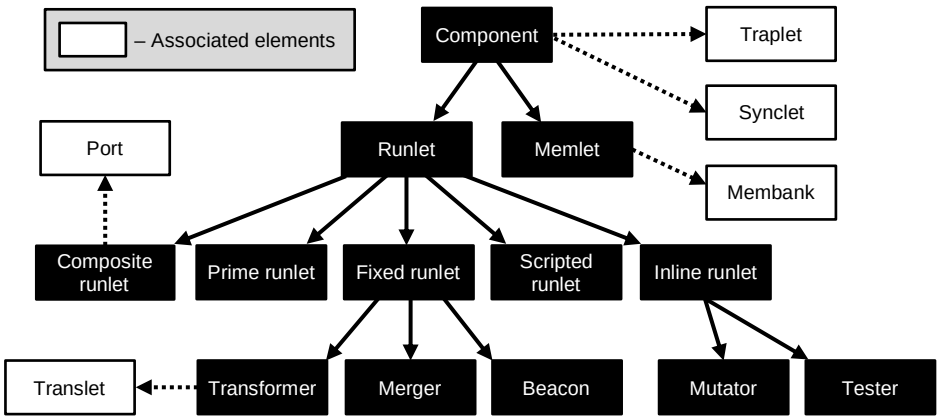


Figure 2. Component taxonomy.

2.1.2.2. Runlets

2.1.2.2.1. Runlet Types

Runlet represents a data processing unit. It may have any number of inputs and outputs (except inline runlets that have a fixed number of inputs and outputs – see "Inline runlet"). The following runlet types are defined in Pipe (see Figure 2):

- Fixed runlets.
- Prime runlets.
- Composite runlets.
- Scripted runlets.
- Inline runlets.

Fixed runlets have predefined behavior as native built-in elements of the language. They are described in 2.1.2.2.2 "Fixed Runlets".

Prime runlets contain executable assets created using any text-based programming language (except scripting programming language Python) called a **prime language**. The source code used to create executable assets inside of a prime runlet is called a **prime source code**.

Prime runlet development would normally happen outside of Pipe development environment because Pipe implementations do not have to provide integrated support of any prime language except allowing import and usage of prime runlets already containing executable assets inside.

Prime runlet is the most difficult type of runlets to create, as it usually requires professional software development skills. However, prime runlets provide many advantages such as better performance and access to a large variety of system and low-level APIs (network, multithreading, etc.).

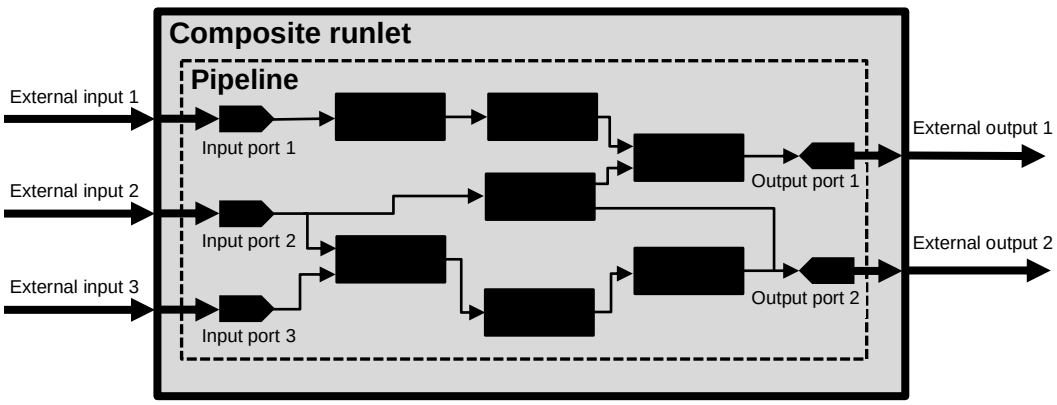


Figure 3. Input and output ports.

Composite runlet contains a pipeline inside. It also contains **input ports** and **output ports** (called collectively **ports**) – connection points inside of a composite runlet for connecting internal components with external inputs and outputs of the composite

General-Purpose Visual Programming Language

Pipe

Large language models are increasingly capable to generate high-quality software code. This poses a question about the role and importance of software developers in near future.

Despite all progress, AI is still not capable to fully comprehend complexity of business domain and technical constraints dictated by a set of specific circumstances. It means AI needs detailed and exhaustive specifications to be submitted for successful code generation. Preparation of such specifications is a difficult task, especially for complex systems. It creates an opportunity for software developers to move from the role of code creators to positions of higher-level designers and system architects by creating software design specifications that will be used as instructions for AI code generation. However, even high-quality and very detailed design does not guarantee AI code generation equivalent to the specifications, because no design can provide comprehensive and unambiguous description of required functionality. The gap between requirements and generated code widens as design complexity grows. As a result, the only way to successfully leverage power of AI for software construction is generating code only for base-level components easily explainable to AI, completing the rest of application via manual coding. However, getting back to manual programming undermines the goal of using AI to eliminate the need for human coding. This is where visual programming language capable to integrate non-visual code fragments into visual workflows is going to be especially useful. This book introduces a new visual programming language "Pipe", where visual blocks can encapsulate text-based code. Pipe allows developers to specify high-level software design via visual workflows, leaving low-level aspects of implementation to AI code generation tools.

As a next step of Pipe evolution, AI will be able to generate Pipe diagrams directly. Visual artefacts are much easier for humans to read and comprehend than text-based code and therefore, automatic Pipe diagram generation will allow more efficient control of an output produced by AI.

ISBN 978-1-0696277-0-4