WETWARE ENGINEERING: APPLYING SOFTWARE ENGINEERING PARADIGMS TO BIOLOGICAL SYSTEM CONSTRUCTION

A PREPRINT

123olp
tukuai.ai@gmail.com

December 29, 2025

ABSTRACT

Software engineering underwent a paradigm shift from monolithic, handcrafted programs to modular, composable systems over five decades—a transformation enabled by standardized interfaces, package managers, version control, and runtime orchestration. Biological engineering, despite remarkable advances in synthetic biology, organoids, and tissue engineering, remains trapped in an analogous "pre-modular" era: each biological system is constructed from scratch, results are difficult to reproduce across laboratories, and there exists no universal language for describing biological component composition.

We propose **Wetware Engineering**, a cross-disciplinary methodology that systematically transfers software engineering's core abstractions—modularity, interface standardization, dependency management, and runtime orchestration—to biological system construction. This is not merely applying computational tools to biology, but fundamentally reconceptualizing how living systems should be designed, described, and assembled.

Our contribution is threefold: (1) **Conceptual Framework**: We define the Component-Interface-Runtime triad as the foundational abstraction for modular biological systems, drawing explicit parallels to software architecture patterns. (2) **Technical Specifications**: We propose Bio-Component Spec, a standardized schema for describing biological modules, and Bio-DSL, a domain-specific language for declarative system composition—both designed following software engineering best practices. (3) **Paradigm Analysis**: We systematically analyze how software engineering concepts map to biological contexts, identifying both direct translations and fundamental differences requiring novel solutions.

Wetware Engineering represents a paradigm-level contribution: shifting biological system construction from "artisanal replication" to "engineered composition." While implementation faces significant biological challenges, establishing this conceptual and methodological foundation is a necessary first step toward reproducible, iterable, and collaborative biological system development.

Keywords Software Engineering · Biological Systems · Cross-Disciplinary Methodology · Modular Design · Domain-Specific Language · Systems Biology · Paradigm Transfer

1 Introduction: The Case for Paradigm Transfer

1.1 Software Engineering's Modular Revolution

The history of software engineering is fundamentally a history of rising abstraction levels. In the 1950s, programmers wrote machine code—sequences of binary instructions tied to specific hardware. The introduction of assembly language provided the first abstraction: human-readable mnemonics replacing numeric opcodes. Structured programming in the 1960s abstracted control flow. Object-oriented programming in the 1980s encapsulated data and behavior together. Component-based development in the 1990s enabled binary-level reuse. Service-oriented architecture in the 2000s abstracted deployment locations. Microservices in the 2010s achieved independent deployment and elastic scaling.

Each abstraction level brought transformative benefits:

Table 1: Evolution of Software Engineering Abstractions

Era	Abstraction	Key Innovation	Impact
1950s	Machine code → Assembly	Human-readable instructions	10x productivity
1960s	Procedures	Structured programming	Manageable complexity
1970s	Modules	Information hiding, interfaces	Team collaboration
1980s	Objects	Data + behavior encapsulation	Reusable libraries
1990s	Components	Binary reuse (COM, JavaBeans)	Third-party ecosystems
2000s	Services	Network-based composition	Enterprise integration
2010s	Microservices	Independent deployment	Cloud-native scalability

The critical insight is that each abstraction level did not merely add convenience—it fundamentally changed what was possible. Before package managers like npm and pip, sharing code meant copying files and manually resolving dependencies. Before containerization, "it works on my machine" was an unsolvable problem.

Today, a software developer can declare `import tensorflow` and instantly access millions of lines of tested, documented, version-controlled code. This is not magic—it is the accumulated result of decades of standardization, tooling, and community building.

1.2 Biological Engineering's "Pre-Modular" State

Biological engineering in 2025, despite extraordinary advances, remains in a state analogous to software engineering circa 1970. Consider the following comparison:

Table 2: Software vs. Biological Engineering: Current State

Software Engineering Concept	Current State in Biology	The Gap
Standard Library	None	Each lab builds from scratch
Package Manager (npm, pip)	None	Cannot declare dependencies
Version Control (git)	None	"This batch differs from last batch"
API Documentation	None	"Ask the original author"
Unit Testing	None	"How long will it last? Maybe a week"
CI/CD Pipeline	None	No automated validation
Containerization (Docker)	None	Environments not reproducible

When a tissue engineer wants to combine a muscle actuator with a neural controller, they face challenges that software engineers solved decades ago:

1. **No standard interfaces**: The muscle was developed in Lab A with specific culture conditions; the neural tissue in Lab B with different protocols. There is no guarantee they can physically or biochemically connect.
2. **No dependency declaration**: What exactly does the muscle need? Glucose concentration? Oxygen levels? Stimulation frequency? This information exists in lab notebooks, not machine-readable specifications.
3. **No version compatibility**: Lab A improved their muscle protocol last month. Does it still work with Lab B's neural tissue? No one knows without re-running experiments.
4. **No composition language**: How do you describe "connect muscle output to sensor input, with closed-loop feedback control"? In natural language, buried in a methods section.

The fundamental problem is conceptual: biological systems are treated as **indivisible wholes** rather than **composable collections of modules**.

1.3 Why Paradigm Transfer, Not Just Tool Application

Existing "computational biology" primarily means:

- Using computers to **analyze** biological data (bioinformatics)

- Using algorithms to **simulate** biological processes (systems biology)
- Using software to **control** biological experiments (lab automation)

These are valuable but insufficient. They apply software as a tool to biology, without changing how biology itself is engineered.

We propose something fundamentally different:

> **Using software engineering's design philosophy to reconceptualize how biological systems are constructed.**

The distinction matters. Tools and methods operate within existing paradigms. Paradigm transfer creates new possibilities that were previously inconceivable.

1.4 Contributions and Paper Structure

This paper makes the following contributions:

1. **Paradigm Definition**: We systematically propose transferring software engineering's core paradigms to biological system construction.
2. **Abstraction Framework**: We define the Component-Interface-Runtime triad as the foundational abstraction for modular biological systems.
3. **Technical Specifications**: We propose Bio-Component Spec v0.1 and Bio-DSL for declarative system composition.
4. **Mapping Analysis**: We systematically analyze how software engineering concepts translate to biological contexts.
5. **Difference Identification**: We identify fundamental differences requiring innovative approaches beyond direct paradigm transfer.

The paper is structured as follows: §2 defines core abstractions; §3 presents systematic mappings; §4 details Bio-Component Specification; §5 describes Bio-DSL; §6 analyzes fundamental differences; §7 positions our work; §8 concludes.

2 Core Abstractions: The Component-Interface-Runtime Triad

2.1 Abstraction as the Essence of Engineering

Edsger Dijkstra observed: "The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise." This insight captures why abstraction is not merely a convenience but the essence of engineering progress.

Software engineering's success stems from identifying **correct abstraction boundaries**:

- Functions abstract instruction sequences
- Objects abstract data and behavior
- Interfaces abstract implementation details
- Services abstract deployment locations
- Containers abstract operating environments

The central question for biological engineering is: **What are the correct abstraction boundaries for living systems?**

We propose that the answer lies in the same triad that revolutionized software: **Component**, **Interface**, and **Runtime**.

2.2 Component: The Unit of Biological Reuse

A **Bio-Component** is a biological unit that:

- Can exist independently (with appropriate life support)

- Can receive energy and nutrients (powerable)
- Can respond to control signals (controllable)
- Can produce functional outputs (functional)
- Can report its state (observable)

This definition deliberately parallels software component definitions.

Table 3: Component Typology

Type	Software Analogy	Biological Examples
Actuator	Output device driver	Muscle, gland, ciliated epithelium
Sensor	Input device driver	Photoreceptor, mechanoreceptor
Processor	CPU, logic unit	Ganglion, brain organoid
Storage	Memory, database	Adipose tissue, bone marrow
Connector	Network interface	Blood vessel, nerve fiber
Metabolic	Power supply	Liver tissue, mitochondria-rich cells

2.3 Interface: The Contract for Composition

The Gang of Four's design principle states: "Program to an interface, not an implementation." This principle enabled the explosion of software reuse.

Biological interfaces are more complex than software interfaces because they operate across multiple physical dimensions simultaneously. We identify four primary dimensions:

1. **Power Interface**: How energy and nutrients flow between components
2. **Signal Interface**: How information is exchanged (electrical, chemical, mechanical, optical)
3. **Isolation Interface**: How components are protected from each other
4. **Mechanical Interface**: How physical forces are transmitted

2.4 Runtime: The Orchestration Layer

A Bio-Runtime must handle responsibilities analogous to software runtimes:

Table 4: Runtime Responsibilities Mapping

Software Runtime	Bio-Runtime
Memory allocation	Nutrient allocation
CPU scheduling	Activity timing control
Network I/O	Signal routing
Health checks	Viability monitoring
Auto-restart	Regeneration/replacement triggering
Logging	Biomarker time-series recording
Load balancing	Workload distribution
Fault isolation	Containing necrosis, inflammation

3 Systematic Mapping: Software → Biological Engineering

3.1 Mapping Framework

Not all software engineering concepts transfer equally to biology. We propose a three-category framework:

Table 5: Mapping Categories

Mapping Type	Definition	Transfer Difficulty
Direct	Concept transfers with minimal adaptation	Low
Analogous	Core idea transfers but requires adaptation	Medium
Novel	No software equivalent; requires new solutions	High

3.2 Direct Mappings

Semantic Versioning: Software's SemVer specification (MAJOR.MINOR.PATCH) transfers directly to Bio-Components:

- MAJOR: Interface-incompatible changes
- MINOR: Backward-compatible enhancements
- PATCH: Optimizations without interface changes

Dependency Declaration: Package manifests can use identical syntax for biological dependencies.

Documentation Standards: README files, API documentation, and usage examples transfer directly.

3.3 Analogous Mappings

Testing → Validation: Key differences include:

- Software tests are deterministic; biological tests are statistical
- Software tests run in milliseconds; biological tests take days/weeks
- Software tests are automated; biological tests require manual intervention

Error Handling → Failure Mode Management: Biological systems have analogous categories:

- Recoverable Degradation (fatigue)
- Irreversible Damage (necrosis)
- Systemic Risk (inflammation, infection)

3.4 Novel Challenges

These challenges have no direct software equivalent:

1. **Immune Compatibility**: Software components do not "reject" each other.
2. **Signal Crosstalk**: Biological components share chemical environments.
3. **Metabolic Coupling**: Components share resources and produce waste.
4. **Living Degradation**: Biological components inherently degrade over time.
5. **Ethical Constraints**: Biological components raise ethical considerations.

4 Bio-Component Specification: Design Rationale

4.1 Design Principles

The Bio-Component Specification draws from established software engineering principles:

SOLID Principles Applied:

- Single Responsibility: A component should perform one biological function
- Open/Closed: Components can be enhanced without changing interfaces
- Liskov Substitution: Compatible components must be interchangeable
- Interface Segregation: Fine-grained interface definitions
- Dependency Inversion: Depend on interface specs, not specific implementations

4.2 Specification Structure

The complete schema includes:

- Identification (id, name, version, license, authors)
- Classification (type, domain, tags)
- Biological Source (organism, tissue type, cell types)
- Interface Definition (inputs, outputs)
- Environmental Requirements (physical, chemical, biological)
- Performance Characteristics (functional, reliability, resources)
- Failure Modes
- Testing specifications
- Dependencies

4.3 Versioning Strategy

We adopt SemVer with biological interpretations:

MAJOR version: Interface-breaking changes (input/output port type changes, required parameter additions)

MINOR version: Backward-compatible additions (new optional output ports, performance improvements)

PATCH version: Backward-compatible fixes (protocol optimizations, documentation updates)

Extended version format for biological specificity:

```
{version}+{batch}.{donor}.{modification}
Example: 2.3.1+batch20251228.donor42.wildtype
```

5 Bio-DSL: Language Design Rationale

5.1 Why a Domain-Specific Language?

Martin Fowler defines a domain-specific language (DSL) as “a computer language specialized to a particular application domain.” DSLs trade generality for expressiveness within their domain.

Benefits of DSLs for biological systems:

- **Expressiveness**: Describe complex assemblies concisely
- **Readability**: Biologists can understand system descriptions
- **Validation**: Catch interface mismatches at “compile time”
- **Abstraction**: Focus on what, not how

5.2 Design Goals

1. **Declarative**: Describe *what* the system is, not *how* to build it
2. **Readable**: A biologist should understand without programming background
3. **Verifiable**: Static analysis can catch errors before physical assembly
4. **Executable**: Can generate runtime configurations
5. **Composable**: Systems can be nested and reused

5.3 Language Constructs

Component Declaration:

```
COMPONENT flexor FROM "muscle-actuator-human-skeletal@^2.3"
COMPONENT sensor FROM "piezo-force-sensor@~1.1" AS force_sensor
```

Connection Declaration:

```
CONNECT sensor.output TO controller.input
CONNECT controller.output TO flexor.stimulation VIA signal_converter
```

Runtime Configuration:

```
RUNTIME {
  perfusion: { temperature: 37C, pH: 7.4, flow_rate: 0.5 mL/min },
  control: { mode: "closed_loop", algorithm: "PID" },
  monitoring: { log_interval: 10s, metrics: ["force", "viability"] }
}
```

Behavioral Logic:

```
ON STARTUP DO {
  SET perfusion.flow_rate = 0.5 mL/min
  WAIT 300s
  SET controller.mode = "active"
}

WHEN flexor.fatigue_index > 0.3 THEN {
  LOG "Fatigue detected"
  REDUCE flexor.stimulation_frequency BY 20%
}
```

5.4 Comparison with Related Languages

Table 6: Comparison with Existing Biological Languages

Language	Abstraction Level	Purpose	Relationship
SBOL	Genetic	DNA sequence description	Lower level
SBML	Molecular	Biochemical networks	Lower level
CellML	Cellular	Cell mathematical models	Lower level
Bio-DSL	Organ/System	Component composition	Higher level

Bio-DSL is designed to **complement** these languages, not replace them.

6 Fundamental Differences and Open Challenges

6.1 Determinism vs. Stochasticity

A software function `add(2, 3)` will always return `5`. A biological muscle stimulated with identical parameters will produce slightly different force each time.

Implications: Interface contracts must be probabilistic; testing must be statistical; runtime must handle uncertainty.

6.2 Discrete vs. Continuous

Software state transitions are instantaneous. Biological state transitions are gradual.

Implications: Interface parameters need tolerance ranges; state definitions need thresholds; timing specifications need ranges.

6.3 Isolation vs. Coupling

Software processes are isolated by the operating system. Biological components share culture medium.

Implications: Explicit coupling declarations; isolation adapter specifications; system-level resource budgeting.

6.4 The Immune System: No Software Equivalent

Software components do not “reject” each other. Biological components from different genetic backgrounds may trigger immune responses.

Required Innovations: Immune compatibility scoring; isolation barrier specifications; compatibility checking in Bio-DSL.

6.5 Living Degradation

Software does not age. Biological components inherently degrade over time.

Required Innovations: Degradation modeling; maintenance protocols; replacement strategies.

6.6 Ethical Constraints

Software has no inherent ethical status. Biological components raise ethical considerations.

Required Innovations: Ethical metadata; capability limits in Bio-DSL.

7 Related Work and Positioning

7.1 Synthetic Biology and Standardization

BioBricks and iGEM pioneered biological standardization at the genetic level. SBOL provides standardized data formats for genetic designs.

Relationship: BioBricks/SBOL operate at the genetic level; Wetware Engineering operates at the organ/system level. They are complementary.

7.2 Organ-on-Chip and Organoids

Organ-on-chip devices and organoids provide physical implementations of Bio-Components. Current systems lack standardized interfaces.

Relationship: Wetware Engineering provides the abstraction framework they need.

7.3 Systems Biology Modeling

SBML and CellML describe how components behave internally (simulation). Bio-DSL describes how components connect externally (composition).

7.4 Positioning Summary

Wetware Engineering’s unique contribution: Providing the **system-level abstraction layer** that connects molecular/-genetic engineering to functional biological systems, using software engineering principles.

8 Conclusion and Future Directions

8.1 Summary of Contributions

This paper has proposed **Wetware Engineering**, a cross-disciplinary methodology that systematically transfers software engineering paradigms to biological system construction.

Conceptual Framework: We defined the Component-Interface-Runtime triad as the foundational abstraction.

Technical Specifications: We proposed Bio-Component Spec v0.1 and Bio-DSL.

Systematic Analysis: We provided mappings categorized as Direct, Analogous, or Novel.

Honest Assessment: We identified fundamental differences that require innovation beyond paradigm transfer.

8.2 The Path Forward

Phase 1 (1-3 years): Refine specifications, develop proof-of-concept tooling, document existing systems.

Phase 2 (3-7 years): Physically implement systems, validate reproducibility, build component registry.

Phase 3 (7-15 years): Establish community governance, commercial adoption, international standardization.

8.3 Closing Thoughts

Software engineering transformed from craft to discipline over five decades. Biological engineering stands at a similar inflection point.

The tools of software engineering—abstraction, modularity, standardization, composition—are not specific to silicon. They are **general principles of managing complexity**. Biology is complex. These principles can help.

The future of biological engineering is modular. The question is: will we design that future deliberately, or stumble into it accidentally?

We choose to design.

Acknowledgments

[To be added]

Data Availability

All specifications and examples are available at: `https://github.com/tukuaiai/wetware-engineering`

References