SYSTEM DESIGN DOCUMENT

QuantumCube Bug Simulator

System Design Document

Version Alpha 0.0001

Hexapod Architecture, Three-Fold Mapping,

and Happy Feet Interaction Model

Author: Chas

Location: Auburn, Alabama

Date: April 2026

"Every bug has six legs. No bugs are harmed. One click, happy feet."

Table of Contents

1. Executive Summary 3

2. System Architecture Overview 4

2.1 Four-Pane IDE Layout

2.2 Data Flow Pipeline

2.3 Component Inventory

3. CLFL Source Language Specification 6

3.1 Keywords and Syntax Categories

3.2 Grammar Structure

3.3 Example Programs

3.4 Syntax Error to Bug Type Mapping

4. Machine Code ISA (Instruction Set Architecture) 8

4.1 32-Bit Instruction Word Layout

4.2 Opcode Families

4.3 Full Opcode Table

4.4 Machine Word Fault Encoding

5. Three-Fold Bug Mapping System 10

5.1 The Three Folds

5.2 H_6 Leak (Fly)

5.3 Phase Desync (Moth)

5.4 Holon Collapse (Ant)

5.5 Matrix Singularity (Beetle)

5.6 Freq Drift (Mosquito)

5.7 Zone Glitch (Cricket)

6. Hexapod Creature Design 13

6.1 Universal Hexapod Anatomy

6.2 Species-Specific Attributes

6.3 Color Palette

6.4 Movement and Animation

7. Happy Feet Interaction Model 15

7.1 Design Philosophy

7.2 Interaction Sequence

7.3 Timing Specification

7.4 Vocabulary and Button Labels

8. Bug Game ↔ Bug Detector Pipeline 17

8.1 Bidirectional Architecture

8.2 BugGameDetectorBridge Design

8.3 Marry Views Feature

8.4 Game Lifecycle

9. C# Companion Architecture (BugsView.cs) 19

9.1 Namespace and Enumerations

9.2 Core Classes

9.3 BugSyntaxAnalyzer and Strategies

9.4 Design Principles

Appendix A: Quick Reference Card 20

A.1 Bug Type Quick Reference Table

A.2 Happy Feet Timing Cheat Sheet

A.3 Button Label Reference

---

1. Executive Summary

The **QuantumCube Bug Simulator** is a pedagogical and ritual debugging tool that visualizes software bugs as six-legged hexapod creatures across three layers of computation. It is designed to transform the traditionally adversarial act of debugging into a humane, joyful, and educational experience.

The simulator operates across **three folds** of representation:

1. **CLFL Source Code (Syntax & Grammar)** — The textual Cube Layer Flow Language where bugs originate as syntax patterns, grammar violations, or semantic anomalies.
2. **Machine Code (ISA-Level Hex Words)** — The compiled instruction set architecture encoding where faults manifest as incorrect immediate values, truncated fields, or duplicate encodings.
3. **Hardware Description Constructs (VHDL & Verilog)** — The hardware description layer where faults appear as stuck signals, zero determinants, identical generics, or truncated parameters.

Central to the system is the **Happy Feet** interaction model: bugs are never harmed, killed, squished, or terminated. Instead, when a user clicks a bug, it performs a joyful tap-dance, scurries to the nearest canvas edge, and fades away with a sparkle effect. The corresponding source line is cleaned automatically, and the system recompiles. The vocabulary is entirely positive — bugs are *invited*, they *dance away*, and they leave *happy*.

The system comprises two primary views — the **Bug Detector** (syntax-driven analysis) and the **Bug Game** (continuous spawning simulation) — connected by a bidirectional synchronization bridge. An optional **Marry Views** overlay allows both canvases to be superimposed at 50% opacity for combined inspection.

Six hexapod species represent six fault types: Fly (H_6 Leak), Moth (Phase Desync), Ant (Holon Collapse), Beetle (Matrix Singularity), Mosquito (Freq Drift), and Cricket (Zone Glitch). Each species has a distinct visual design, color, wing configuration, and movement pattern, while all share the universal hexapod body plan of six legs arranged in three pairs.

Document Scope

This document covers system architecture, language specification, ISA encoding, three-fold mapping, creature design, interaction model, pipeline architecture, and companion C# class hierarchy for Version Alpha 0.0001.

2. System Architecture Overview

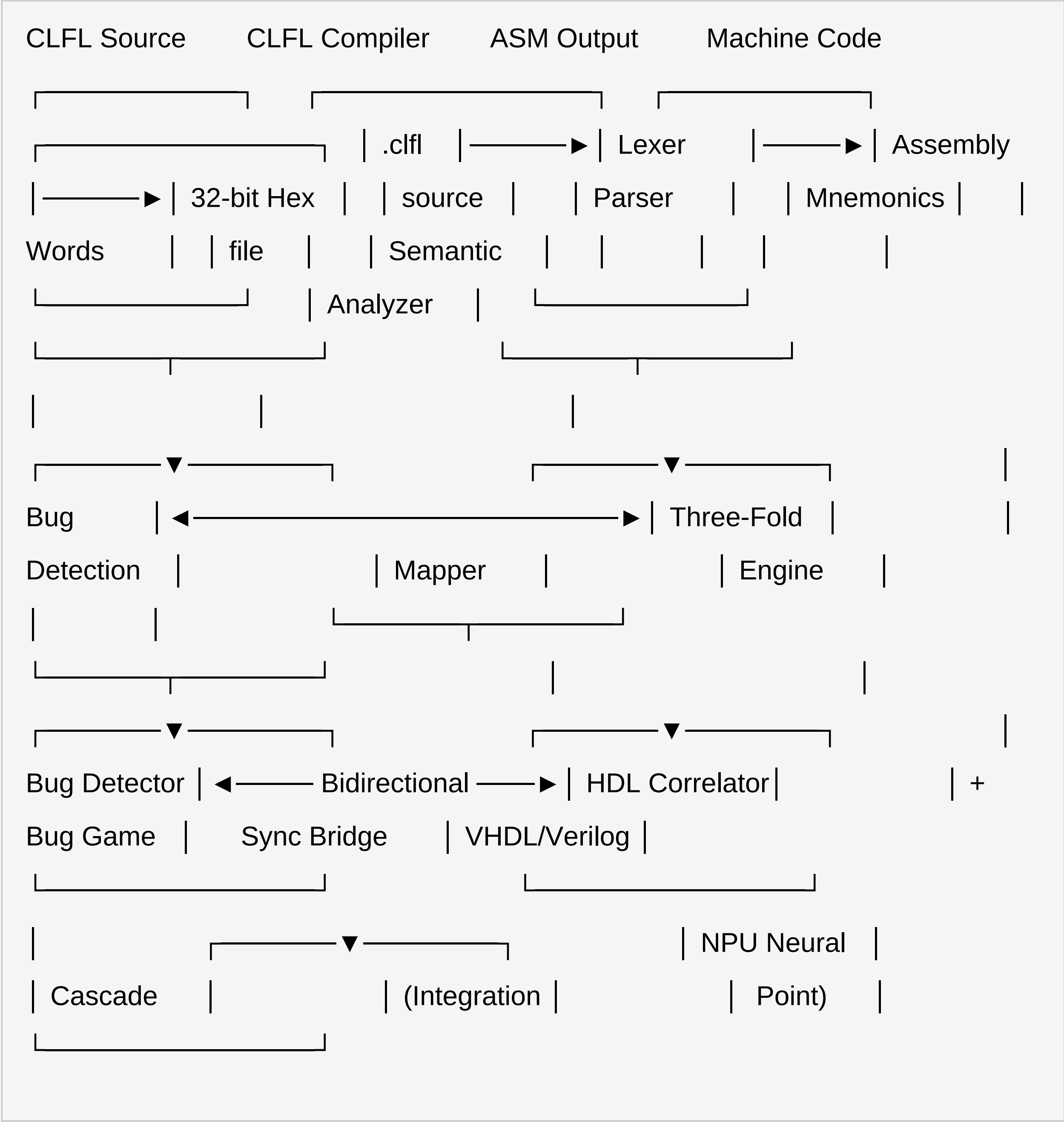
2.1 Four-Pane IDE Layout

The QuantumCube Bug Simulator presents a four-pane integrated development environment. Each pane serves a distinct role in the debugging workflow, and all four are synchronized through the central compiler pipeline and event bus.

```
+---------------------------------------------------------------------+ |
QuantumCube Bug Simulator IDE                    | +----------------------------------
+---------------------------------+ |                          |                          || PANE
1: Source Editor          | PANE 2: ASM / Machine Code      ||
|                          ||  ┌────────────────────────────────────────┐  |
┌────────────────────────────────────────┐  || │ LOOP Delta 0            │  |
│ 0xA0B00000  LOOP_INIT      │  || │ FLOW Alpha->Alpha->Alpha  │  | │
0x41601004  LAYER_BIND     │  || │ INIT holon [0,0,0,0]       │  | │ 0x11000000
REG_SET R0      │  || │ XFORM holon T1            │  | │ 0x11100000  REG_SET
R1      │  || │ FREQ 3.14159 GHz          │  | │ 0x2040000A  FREQ_LOAD      │  |
| │ FLOW Delta->Theta->Alpha  │  | │ 0x41600001  LAYER_BIND      │  ||
└────────────────────────────────────────┘  |
└────────────────────────────────────────┘  ||                          |
| +---------------------------------+---------------------------------+ |                          |
|| PANE 3: Bug Detector + Bug Game  |  PANE 4: QuantumCube Visual      ||
|                          ||  ┌────────────────────────────────────────┐  |
┌────────────────────────────────────────┐  || │ 🪰 Fly    → Line 1       │  |
│                         │  || │ 🦋 Moth   → Line 2       │  | │   ╔═════╦═════╦═════╗
│  || │ 🐜 Ant    → Line 3       │  | │    ║ α ║ β ║ θ ║          │  || │ 🪲 Beetle →
Line 4       │  | │    ╠═════╬═════╬═════╣          │  || │ 🦟 Mosquito → Line 5      │
| │    ║ δ ║ φ ║ γ ║          │  || │ 🦗 Cricket → Line 6      │  | │
╚═════╩═════╩═════╝          │  ||  └────────────────────────────────────────┘
|  └────────────────────────────────────────┘  ||                          |
| +---------------------------------+---------------------------------+
```

2.2 Data Flow Pipeline

The compiler pipeline transforms CLFL source into machine code, then correlates each compiled instruction with its hardware description equivalent. Bug detection runs in parallel across all three folds.



The **NPU Neural Cascade** integration point represents a future extension where neural processing unit hardware accelerates pattern matching for bug detection

across all three folds simultaneously. In Version Alpha 0.0001, this is a stub interface.

2.3 Component Inventory

Subsystem	Component	Description	Status
Source Editor	CLFL Editor	Syntax-highlighted editor for Cube Layer Flow Language with live error markers	Active
Compiler	CLFL Lexer	Tokenizer for CLFL keywords, literals, identifiers, operators, delimiters	Active
	CLFL Parser	Statement-level parser producing an abstract syntax tree	Active
	Code Generator	Emits 32-bit machine words from parsed AST nodes	Active
Machine Code	ISA Encoder	Packs opcode, sub-opcode, register, modifier, and immediate into 32-bit words	Active
	Disassembler	Decodes machine words back to mnemonic form for display in Pane 2	Active
Bug Detection	Bug Detector	Syntax-driven analysis engine identifying fault patterns across all three folds	Active
	Bug Game	Continuous bug spawning simulation with interactive Happy Feet mechanics	Active
	BugGameDetectorBridge	Bidirectional synchronization between Bug Game and Bug Detector views	Active

Subsystem	Component	Description	Status
Three-Fold	ThreeFoldMapper	Maps each bug type across CLFL source, machine word, and HDL construct	Active
	HDL Correlator	Generates VHDL entities/signals and Verilog modules/registers for each fault	Active
Visualization	QuantumCube Renderer	3D cube visualization showing layer states and frequency mappings	Active
	Hexapod Renderer	Draws six-legged creatures with species-specific anatomy and animation	Active
	Marry Views Overlay	50% opacity overlay merging Bug Detector and Bug Game canvases	Active
C# Companion	BugsView.cs	Complete class hierarchy: enums, models, views, analyzer, bridge, factory	Active
NPU Integration	Neural Cascade	Hardware-accelerated bug pattern detection (future integration stub)	Planned

3. CLFL Source Language Specification

The **Cube Layer Flow Language (CLFL)** is a domain-specific language designed for specifying cube layer operations, frequency bindings, holon transformations, and data flow within the QuantumCube system. CLFL programs are compiled to machine code and simultaneously correlated with VHDL/Verilog hardware description constructs.

3.1 Keywords and Syntax Categories

CLFL defines ten reserved keywords and seven syntax categories. The lexer tokenizes source text into these categories before parsing.

Keyword	Description	Usage Pattern
`LOOP`	Defines an iteration block with a named layer and count	`LOOP <layer> <count>`
`FLOW`	Establishes data flow between named layers using arrow notation	`FLOW <A>-><B>-><C>`
`INIT`	Initializes a named object with a value vector	`INIT <obj> [v0,v1,...]`
`XFORM`	Applies a named transformation matrix to an object	`XFORM <obj> <matrix>`
`FREQ`	Sets a frequency value with unit specifier	`FREQ <value> <unit>`
`ZONE`	Declares a named spatial or logical zone	`ZONE <name> <params>`
`LAYER`	Defines a cube layer with index and properties	`LAYER <name> <idx>`
`HOLON`	Declares a holon object with component vector	`HOLON <name> <dim>`
`EMIT`	Outputs a signal or value to the visualization pipeline	`EMIT <signal> <val>`
`SYNC`	Synchronizes named layers to a common clock boundary	`SYNC <layer1> <layer2>`

Syntax Categories

Category	Description	Examples
Keyword	Reserved language keywords	`LOOP`, `FLOW`, `INIT`
Literal	Numeric and string constants	`0`, `3.14159`, `0.0`
Identifier	Layer names, object names,	`Delta`, `holon`, `T1`

Category	Description	Examples
	matrix names	
Operator	Flow and assignment operators	`->`, `=`
Delimiter	Brackets and commas for vector notation	`[`, `]`, `,`
Directive	Unit specifiers and mode flags	`GHz`, `MHz`
Comment	Line comments beginning with double-slash	`// this is a comment`

3.2 Grammar Structure

CLFL uses a statement-based grammar where each line is a self-contained command. There are no block-level constructs, nested scopes, or multi-line expressions. Each statement begins with a keyword and is followed by ordered parameters.

Grammar Railroad (Text Representation):

```
PROGRAM    ::= STATEMENT*   STATEMENT  ::= KEYWORD PARAMS
NEWLINE   KEYWORD    ::= 'LOOP' | 'FLOW' | 'INIT' | 'XFORM' | 'FREQ'               |
'ZONE' | 'LAYER' | 'HOLON' | 'EMIT' | 'SYNC'    PARAMS     ::= PARAM (' '
PARAM)*   PARAM      ::= IDENTIFIER | LITERAL | VECTOR | FLOW_CHAIN |
DIRECTIVE    IDENTIFIER ::= [A-Za-z][A-Za-z0-9_]*   LITERAL    ::= INTEGER |
FLOAT   INTEGER    ::= [0-9]+   FLOAT      ::= [0-9]+ '.' [0-9]+   VECTOR     ::= '['
LITERAL (',' LITERAL)* ']'   FLOW_CHAIN ::= IDENTIFIER ('->' IDENTIFIER)+
DIRECTIVE  ::= 'GHz' | 'MHz' | 'kHz'   COMMENT    ::= '//' .* NEWLINE
```

3.3 Example Programs

Program 1 — Harmonic Cascade:

```
// Harmonic Cascade: phase-aligned layer sweep LAYER Alpha 0 LAYER Beta 1 LAYER Theta 2 LAYER Delta 3 FREQ 1.0 GHz FLOW Alpha->Beta->Theta->Delta LOOP Alpha 128 SYNC Alpha Beta EMIT cascade_out 1
```

Program 2 — Phase Matrix:

```
// Phase Matrix: holon transformation with matrix multiply HOLON h1 4 INIT h1 [1.0, 0.5, 0.25, 0.125] XFORM h1 T_rotation FREQ 2.5 GHz EMIT phase_out 1
```

Program 3 — Frequency Sweep:

```
// Frequency Sweep: iterate frequencies across all zones ZONE primary 4 FREQ 0.5 GHz LOOP Delta 256 FLOW Delta->Theta->Alpha->Beta SYNC Delta Theta EMIT sweep_result 1
```

3.4 Syntax Error to Bug Type Mapping

Each category of syntax error or semantic anomaly in CLFL maps deterministically to a specific hexapod bug type:

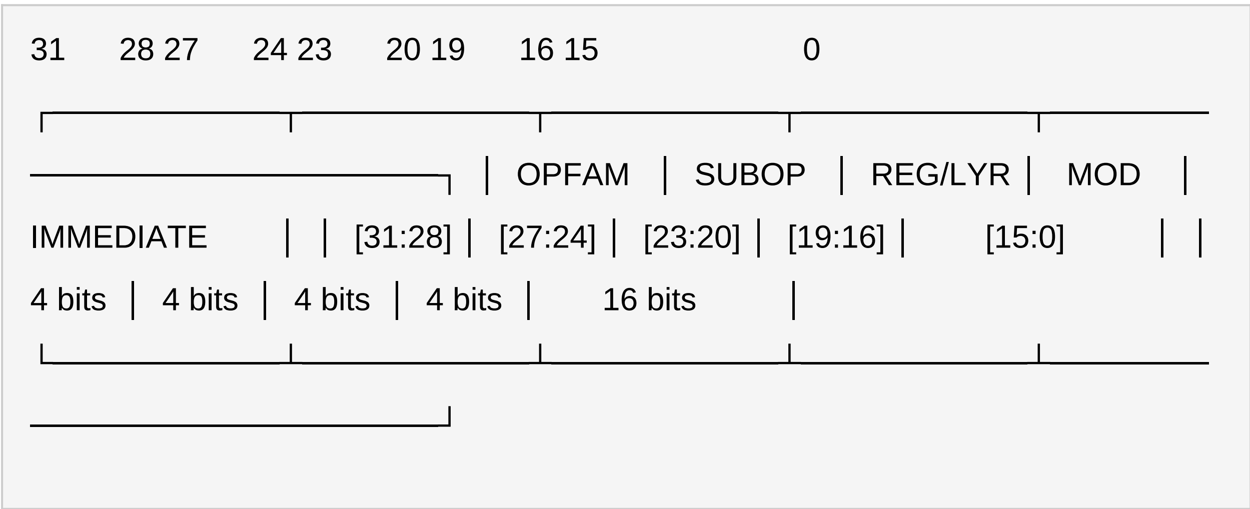
Syntax/Semantic Error	Bug Type	Detection Rule
Zero loop iteration count	Fly (H_6 Leak)	LOOP statement with literal `0` as the count parameter
Identical flow targets	Moth (Phase Desync)	FLOW chain where all identifiers are identical
All-zero initialization vector	Ant (Holon Collapse)	INIT vector where every component equals `0.0`

Syntax/Semantic Error	Bug Type	Detection Rule
Duplicate matrix rows	Beetle (Matrix Singularity)	XFORM references a matrix with linearly dependent rows
Non-integer frequency value	Mosquito (Freq Drift)	FREQ literal with non-zero fractional part
Reversed layer ordering	Cricket (Zone Glitch)	FLOW chain with layers in reverse canonical order (Delta first)

4. Machine Code ISA (Instruction Set Architecture)

The QuantumCube ISA defines a 32-bit fixed-width instruction format. All CLFL source statements compile to one or more machine words. The ISA is designed so that common bug patterns are detectable through bitwise inspection of the encoded fields.

4.1 32-Bit Instruction Word Layout



Field	Bits	Width	Description
OPFAM	[31:28]	4 bits	Opcode family selector: identifies

Field	Bits	Width	Description
			the major operation category
SUBOP	[27:24]	4 bits	Sub-opcode: selects specific operation within the family
REG/LYR	[23:20]	4 bits	Register or layer index (0–15)
MOD	[19:16]	4 bits	Modifier flags: addressing mode, sign extension, precision
IMMEDIATE	[15:0]	16 bits	Immediate value: constants, offsets, counts, or frequency mantissa

4.2 Opcode Families

Family Code	Family Name	Operations
`0x1_`	REG Operations	Register set, register copy, register clear, register exchange
`0x2_`	FREQ Operations	Frequency load, frequency validate, frequency scale, frequency compare
`0x4_`	LAYER Operations	Layer bind, layer unbind, layer swap, layer query
`0x8_`	MATRIX Operations	Matrix-vector multiply, matrix transpose, matrix determinant, matrix inverse
`0xA_`	LOOP Operations	Loop init, loop decrement, loop branch, loop terminate

4.3 Full Opcode Table

Hex Encoding	Mnemonic	Description
`0x11000000`	REG_SET R0	Set register R0 to

Hex Encoding	Mnemonic	Description
		immediate value (imm=0x0000)
`0x11100000`	REG_SET R1	Set register R1 to immediate value (imm=0x0000)
`0x11200000`	REG_SET R2	Set register R2 to immediate value (imm=0x0000)
`0x11300000`	REG_SET R3	Set register R3 to immediate value (imm=0x0000)
`0x2040000A`	FREQ_LOAD	Load frequency value; imm=0x000A (mantissa truncated)
`0x22400000`	FREQ_VALID	Validate frequency against bounds; imm=0x0000 (no bounds set)
`0x41600001`	LAYER_BIND	Bind layer with period value; imm=0x0001 (period=1)
`0x41601004`	LAYER_BIND	Bind layer variant; mod=0x1, imm=0x1004 (period=4, variant encoding)
`0x810C0000`	MAT_VEC_MUL	Matrix-vector multiply; reg=0, mod=0xC (32-bit precision)
`0x811C1000`	MAT_VEC_MUL	Matrix-vector multiply variant; reg=1, mod=0xC, imm=0x1000
`0xA0B00000`	LOOP_INIT	Initialize loop counter; imm=0x0000 (zero iterations)

4.4 Machine Word Fault Encoding

Machine word faults are detectable by inspecting specific bit fields. The following patterns indicate compiled bugs:

Fault Pattern	Detection Rule	Resulting Bug Type
Zero Immediate	imm[15:0] == 0x0000	Fly (H_6 Leak): loop counter is zero; Ant (Holon Collapse): register value is zero
Truncated Mantissa	Fractional source value truncated to integer in 16-bit field	Mosquito (Freq Drift): frequency 3.14159 encoded as 0x000A (truncated)
Duplicate Encodings	Two or more consecutive instructions with identical imm or identical REG/MOD	Moth (Phase Desync): all layer bindings with same period; Beetle (Matrix Singularity): identical matrix rows
Swapped Fields	REG/LYR field contains unexpected layer index for position	Cricket (Zone Glitch): Delta bound to slot 0 instead of expected slot 3

Note: Endianness

All machine words are stored and displayed in big-endian byte order. Hex encodings in this document read left-to-right from MSB (bit 31) to LSB (bit 0).

5. Three-Fold Bug Mapping System

The Three-Fold Bug Mapping System is the **core intellectual contribution** of the QuantumCube Bug Simulator. Every bug type is defined not as a single-layer fault, but as a correlated pattern that manifests across all three layers of computation simultaneously. This section provides the complete specification for each of the six bug types.

5.1 The Three Folds

Fold	Layer Name	Description
1	CLFL Source (Syntax & Grammar)	The textual source code where the bug originates as a syntax pattern, grammar violation, or semantic anomaly
2	Machine Word (ISA Encoding)	The compiled machine code where the fault manifests as incorrect immediate values, truncated fields, or duplicate encodings
3	HDL Construct (VHDL & Verilog)	The hardware description where the fault appears as stuck signals, zero determinants, identical generics, or truncated parameters

---

5.2 H_6 Leak — Fly (Winged Hexapod)

Fold 1: CLFL Source

```
LOOP Delta 0
```

Fault: Zero iterations — the loop body never executes. The `LOOP` keyword is present but the count literal is `0`, making the statement semantically null. It occupies source space and compiles to a machine word but produces no computational effect.

Fold 2: Machine Word

```
0xA0B00000  LOOP_INIT  imm=0x0000
```

Fault: The immediate field [15:0] is `0x0000`, encoding zero iterations. The LOOP_INIT instruction is properly formed — the opcode family (`0xA`), sub-opcode (`0x0`), and register/modifier fields are all valid — but the loop will never execute because the counter starts at zero.

Fold 3: HDL Construct

VHDL:

```
signal loop_ctr : unsigned(31 downto 0) := (others => '0'); signal tick_reg :
unsigned(31 downto 0) := (others => '0');  -- loop_ctr initialized to 0; increment never
fires -- tick_reg stays at initial value indefinitely
```

Verilog:

```
reg [31:0] tick_count = 32'd0;  always @(posedge clk) begin     if (loop_ctr > 0)       //
never true: loop_ctr == 0         tick_count
<= data-id="600" tick_count + 1; end // tick_count stuck at reset value
```

5.3 Phase Desync — Moth (Winged Hexapod)

Fold 1: CLFL Source

```
FLOW Alpha->Alpha->Alpha->Alpha
```

Fault: All flow targets are identical. The `FLOW` statement defines a data path, but every node points to the same layer. There is no phase differentiation, no data movement, and no pipeline progression. The flow graph collapses to a self-loop.

Fold 2: Machine Word

```
0x41601004  LAYER_BIND  reg=6, mod=0x1, imm=0x1004
```

Fault: All layer bindings use the same period value (`4`). When the flow chain compiles, each stage binds to a layer with an identical period, eliminating any phase offset between pipeline stages.

Fold 3: HDL Construct

VHDL:

```
entity phase_pipe is   generic(     G_BETA_PERIOD  : integer := 4;     G_ALPHA_PERIOD : integer := 4;     G_THETA_PERIOD : integer := 4;     G_DELTA_PERIOD : integer := 4   ); end entity phase_pipe; -- All four generics are identical; no phase offset possible
```

Verilog:

```
module phase_pipe #(     parameter BETA_PERIOD  = 4,     parameter ALPHA_PERIOD = 4,     parameter THETA_PERIOD = 4,     parameter DELTA_PERIOD = 4 ) ( ... ); // Phase comparison logic always evaluates true: // (ALPHA_PERIOD == BETA_PERIOD) is always 1'b1
```

5.4 Holon Collapse — Ant (Wingless Hexapod)

Fold 1: CLFL Source

```
INIT holon [0.0, 0.0, 0.0, 0.0]
```

Fault: All four holon components are initialized to zero. The holon object exists but has zero magnitude, making normalization impossible (division by zero) and rendering all subsequent transformations undefined.

Fold 2: Machine Word

```
0x11000000  REG_SET R0  imm=0x0000 0x11100000  REG_SET R1  imm=0x0000
0x11200000  REG_SET R2  imm=0x0000 0x11300000  REG_SET R3  imm=0x0000
```

Fault: Four consecutive `REG_SET` instructions, all with `imm=0x0000`. Each register receives the value zero. The pattern of four consecutive zero-immediate register sets is the machine-level signature of Holon Collapse.

Fold 3: HDL Construct

VHDL:

```
for i in 0 to 3 loop     components(i)
<= data-id="636" to_signed(0, 32); end loop; -- magnitude |H| = sqrt(0^2 + 0^2 + 0^2
+ 0^2) = 0 -- normalization: components(i) / |H| => division by zero
```

Verilog:

```
components[0]

<= data-id="640" 32'sd0; components[1]

<= 32'sd0; components[2]

<= 32'sd0; components[3]

<= 32'sd0;  // sqrt_est = isqrt(0 + 0 + 0 + 0) = 0 // normalize: components[i] / sqrt_est
=> division by zero
```

5.5 Matrix Singularity — Beetle (Wingless Hexapod)

Fold 1: CLFL Source

```
XFORM holon T1
```

Fault: The transformation matrix `T1` contains duplicate rows, making it singular (determinant = 0). The XFORM instruction is syntactically valid, but the referenced matrix is non-invertible, so the transformation loses information irreversibly.

Fold 2: Machine Word

```
0x810C0000  MAT_VEC_MUL  reg=0, mod=0xC, imm=0x0000 0x811C1000
```

```
MAT_VEC_MUL  reg=1, mod=0xC, imm=0x1000
```

Fault: Row 0 dot product equals row 1 dot product because the matrix rows are identical. The two `MAT_VEC_MUL` instructions produce identical outputs despite operating on different registers, revealing the underlying matrix singularity.

Fold 3: HDL Construct

VHDL:

```
signal T : matrix_t(0 to 3)(0 to 3); -- T(0) and T(1) are identical matrix_row_t signals
-- det_reg
<= data-id="657" fp_mul(T(0)(0), T(1)(1)) --          - fp_mul(T(0)(1), T(1)(0)); -- Since
T(0) == T(1): det_reg evaluates to zero (singular)
```

Verilog:

```
// T[0][j] == T[1][j] for all j // Determinant computation: assign det_out = (T[0][0] * T[1]
[1]) - (T[0][1] * T[1][0]); // Since rows are identical: det_out
<= data-id="661" 0 // Matrix is non-invertible
```

5.6 Freq Drift — Mosquito (Winged Hexapod)

Fold 1: CLFL Source

```
FREQ 3.14159 GHz
```

Fault: Non-integer frequency value. The fractional part (`.14159`) cannot be faithfully represented in the 16-bit immediate field of the machine word or in integer-typed HDL generics/parameters. The compiled value silently truncates.

Fold 2: Machine Word

```
0x2040000A  FREQ_LOAD  imm=0x000A
```

Fault: The mantissa is truncated in the 16-bit immediate field. The source value `3.14159` is encoded as `0x000A`, losing all fractional precision. The immediate field can only represent integers, so the fractional component is silently discarded.

Fold 3: HDL Construct

VHDL:

```
entity freq_gen is   generic(     G_FREQ_DEFAULT_GHZ : integer := 3   ); end entity
freq_gen; -- Integer generic type loses the .14159 fractional part entirely --
Frequency error: (3.14159 - 3) / 3.14159 = 4.51%
```

Verilog:

```
module freq_gen #(     parameter FREQ_DEFAULT = 3    // integer truncates pi to 3 )
( ... );  reg [7:0] freq_lat; always @(posedge clk)     freq_lat

<= data-id="682" 8'd3;    // 3.14159 becomes 3 // Frequency drift: 4.51% systematic
```

```
error
```

5.7 Zone Glitch — Cricket (Wingless Hexapod)

Fold 1: CLFL Source

```
FLOW Delta->Theta->Alpha->Beta
```

Fault: Reversed layer ordering. The canonical layer order is Alpha (slot 0) → Beta (slot 1) → Theta (slot 2) → Delta (slot 3). This flow chain places Delta (the innermost layer) at the outermost position (slot 0), inverting the entire pipeline hierarchy.

Fold 2: Machine Word

```
0x41600001  LAYER_BIND  reg=6, mod=0x0, imm=0x0001
```

Fault: Delta is bound to slot 0 instead of slot 3. The period values are swapped: Delta receives `imm=0x0001` (period=1, fastest) when it should receive period=8 (slowest), and Beta receives period=8 (slowest) when it should receive period=1 (fastest).

Fold 3: HDL Construct

VHDL:

```
generic(     G_DELTA_PERIOD : integer := 8;  -- WRONG: should be 1
G_THETA_PERIOD : integer := 4;     G_ALPHA_PERIOD : integer := 2;
G_BETA_PERIOD  : integer := 1   -- WRONG: should be 8 ); -- Delta receives Beta's
period (8 instead of 1) -- Pipeline runs in reverse temporal order
```

Verilog:

```
parameter DELTA_PERIOD = 8,   // WRONG! Should be 1 parameter
THETA_PERIOD = 4, parameter ALPHA_PERIOD = 2, parameter BETA_PERIOD
= 1;   // WRONG! Should be 8 // Layer periods inverted: outer layers run faster than
inner layers // Data arrives before pipeline stages are ready
```

6. Hexapod Creature Design

All bugs in the QuantumCube Bug Simulator are **hexapods** — six-legged arthropod creatures. This is a core design invariant: no bug may have fewer or more than six legs. The hexapod body plan provides visual consistency while species-specific attributes (color, wings, proportions) communicate bug type at a glance.

6.1 Universal Hexapod Anatomy

Every bug species shares the following anatomical structure:

```
ANTENNAE                   /     \                 /       \          ┌──────────────┐            │
HEAD  │  ← smaller oval, two eye dots              └──────┬───────┘          L1 ╱  │   ╲
R1       ← Legs at ±30° from horizontal        L2 ╱ ┌────┴─────┐ ╲ R2    ← Legs at ±60°
from horizontal       L3 ╱  │BODY │   ╲ R3   ← Legs at ±90° from horizontal
```

```
│(oval)│          └──────────┘          [WINGS if applicable — on top of body]
```

Feature	Specification
Body	Oval/elliptical shape, species-specific fill color. Proportions: width × 0.6 = height. Stroke: 1px, color 20% darker than fill.
Head	Smaller oval at front of body (approximately 40% of body width). Contains two circular eye dots (2px radius, black fill).
Antennae	Two thin curved lines extending from the top of the head. Stroke: 1px, same color as body stroke. Curvature: gentle arc outward.
Legs (Total: 6)	Six legs, three per side. Stroke width: 2px, color slightly darker than body fill. Each leg is a two-segment line with a joint at the midpoint.
Left Legs	L1 at −30°, L2 at −60°, L3 at −90° from horizontal center axis
Right Legs	R1 at +30°, R2 at +60°, R3 at +90° from horizontal center axis
Wings	Species-dependent. Present on Fly, Mosquito, and Moth. Absent on Beetle, Ant, and Cricket.

6.2 Species-Specific Attributes

Species	Bug Type	Wings	Distinctive Features
Fly	H_6 Leak	Yes (2)	Two small oval wings on top of body, semi-transparent fill (opacity 0.4). Standard body proportions. Rapid wing shimmer during movement.

Species	Bug Type	Wings	Distinctive Features
Moth	Phase Desync	Yes (2)	Two broader oval wings with subtle pattern overlay. Wings are wider than Fly wings (1.5× body width). Gentle flutter animation.
Mosquito	Freq Drift	Yes (2)	Two thin pointed wings, semi-transparent (opacity 0.3). Elongated body (width × 0.4 = height). Thin proboscis extending from head.
Beetle	Matrix Singularity	No	Hard shell (wider body, width × 0.75 = height). Slightly wider than other species. Shell has a center line detail. No wings visible.
Ant	Holon Collapse	No	Smaller body (80% scale). Pronounced head (50% of body width). Three distinct body segments visible (head, thorax, abdomen). No wings.
Cricket	Zone Glitch	No	Longer back legs (L3/R3 are 1.5× length of other legs) for jumping appearance. Slightly elongated body. No wings.

6.3 Color Palette

Species	Fill Color	Hex Code	Stroke Color	Color Name
Fly		`#66ff66`	`#44cc44`	Bright Green

Species	Fill Color	Hex Code	Stroke Color	Color Name
Beetle		`#8B4513`	`#6B3410`	Saddle Brown
Mosquito		`#aaaaaa`	`#888888`	Gray
Moth		`#DDA0DD`	`#BB80BB`	Plum
Ant		`#222222`	`#111111`	Near-Black
Cricket		`#556B2F`	`#3D4E22`	Dark Olive Green

6.4 Movement and Animation

Random Walk

All hexapods use a random walk movement pattern when in the **Active** state:

- **Speed:** 30–60 pixels per second (species-dependent)
- **Direction changes:** Every 1–3 seconds, a new random heading is selected (0°–360°)
- **Boundary behavior:** When approaching a canvas edge (within 20px), heading reverses toward center
- **Rotation:** Body rotates smoothly to face movement direction (100ms transition)

Alternating Tripod Gait

Leg animation follows the standard arthropod alternating tripod gait pattern:

Phase	Moving Legs	Planted Legs	Duration
Phase A	L1, R2, L3 (legs 1, 4, 5)	R1, L2, R3 (legs 2, 3, 6)	150ms
Phase B	R1, L2, R3 (legs 2, 3, 6)	L1, R2, L3 (legs 1, 4, 5)	150ms

The tripod gait ensures that three legs are always in contact with the surface, providing stability. Moving legs arc upward by 5px during their swing phase and return to the resting angle on contact. Gait cycle: 300ms total (Phase A + Phase B).

Design Principle

The hexapod body plan is biologically inspired but stylized for clarity. At typical rendering sizes (20–40px body width), species must be distinguishable by color and silhouette alone, without requiring fine detail.

7. Happy Feet Interaction Model

7.1 Design Philosophy

> "No bugs are harmed in this simulator. Ever. They simply dance away."

The Happy Feet interaction model represents a fundamental departure from traditional debugging metaphors. In conventional debugging tools, bugs are "killed," "squished," "terminated," or "destroyed." The QuantumCube Bug Simulator **rejects all destructive vocabulary** and replaces it with an entirely positive lexicon rooted in joy, dance, and departure.

This design serves two purposes:

4. **Pedagogical:** By removing the adversarial framing, the simulator encourages curiosity about bug behavior rather than frustration. Users learn to observe, understand, and gently resolve faults.
5. **Ritual:** The Happy Feet sequence transforms debugging from a chore into a small moment of delight. Each bug departure is a micro-celebration.

❗ **Vocabulary Rules (Mandatory)**

NEVER use: kill, squish, terminate, destroy, eliminate, exterminate, crush, smash, swat, zap, nuke, eradicate.
ALWAYS use: invite, dance away, happy feet, danced away, departed, scurried off, farewell, waltzed out, tapped away.

7.2 Interaction Sequence

The interaction trigger is a **single click** (not double-click) on any hexapod in either the Bug Detector or Bug Game canvas. The complete Happy Feet sequence proceeds through four phases:

Phase 1: Dance (600ms)

All six legs enter a rapid alternating wiggle pattern. The left leg group (L1, L3, R2 — legs 1, 5, 4) and right leg group (R1, L2, R3 — legs 2, 3, 6) alternate at 80ms intervals, creating a tap-dance effect. The body remains stationary. A subtle scale pulse (1.0 → 1.05 → 1.0) occurs at the start. The bug's eyes change from black dots to small arcs (smile expression).

Phase 2: Scurry (300ms)

The bug moves quickly toward the nearest canvas edge while legs continue wiggling. The direction is calculated as the vector from the bug's center to the closest edge point. Movement speed increases to 200 pixels/second (3–4× normal walking speed). If the bug is equidistant from two edges, it prefers the bottom edge.

Phase 3: Fade (200ms)

Opacity transitions from 1.0 → 0.0 as the bug reaches the canvas edge. Simultaneously, a brief **stardust sparkle effect** plays: 4–5 white dots (3px radius) expand outward from the bug's last position over 200ms, then fade. The sparkle conveys a sense of magic rather than destruction.

Phase 4: Cleanup

After the fade completes:

- The bug entity is removed from the active collection
- The corresponding CLFL source line fades (text color → background color over 600ms) and is deleted
- An automatic recompile fires, regenerating the machine code and HDL correlation for remaining bugs
- A toast notification appears: **"🦶 Happy feet! Bug danced away. Source cleaned."**

All-Clear State

When no bugs remain in the view after the final Happy Feet departure:

- Display: **"✓ All clear — every bug danced away happy!"**
- The canvas background transitions to a peaceful green glow (`#f0fff0` → `#ffffff` over 2 seconds)
- A single celebratory sparkle burst plays at canvas center

7.3 Timing Specification

Phase	Duration	Cumulative	Key Animation
Dance	600ms	0–600ms	Leg wiggle at 80ms intervals, scale pulse, smile eyes
Scurry	300ms	600–900ms	200px/s toward nearest edge, continued wiggle
Fade	200ms	900–1100ms	Opacity 1.0 → 0.0, stardust sparkle (4–5 dots)
Cleanup	600ms	1100–1700ms	Source line fade, entity removal, recompile, toast
Total	**1,700ms** from click to complete cleanup		

7.4 Vocabulary and Button Labels

All user-facing text adheres to the positive vocabulary system:

Button Label	Action
🐛 **Invite Bug**	Spawns a specific bug type into the Bug Game canvas
🦶 **Happy Feet**	Gives happy feet to a selected bug (triggers the departure sequence)
🎲 **Random Bug**	Spawns a randomly-selected bug type into the canvas
🔍 **Detect Bugs**	Runs the Bug Detector analysis on current CLFL source

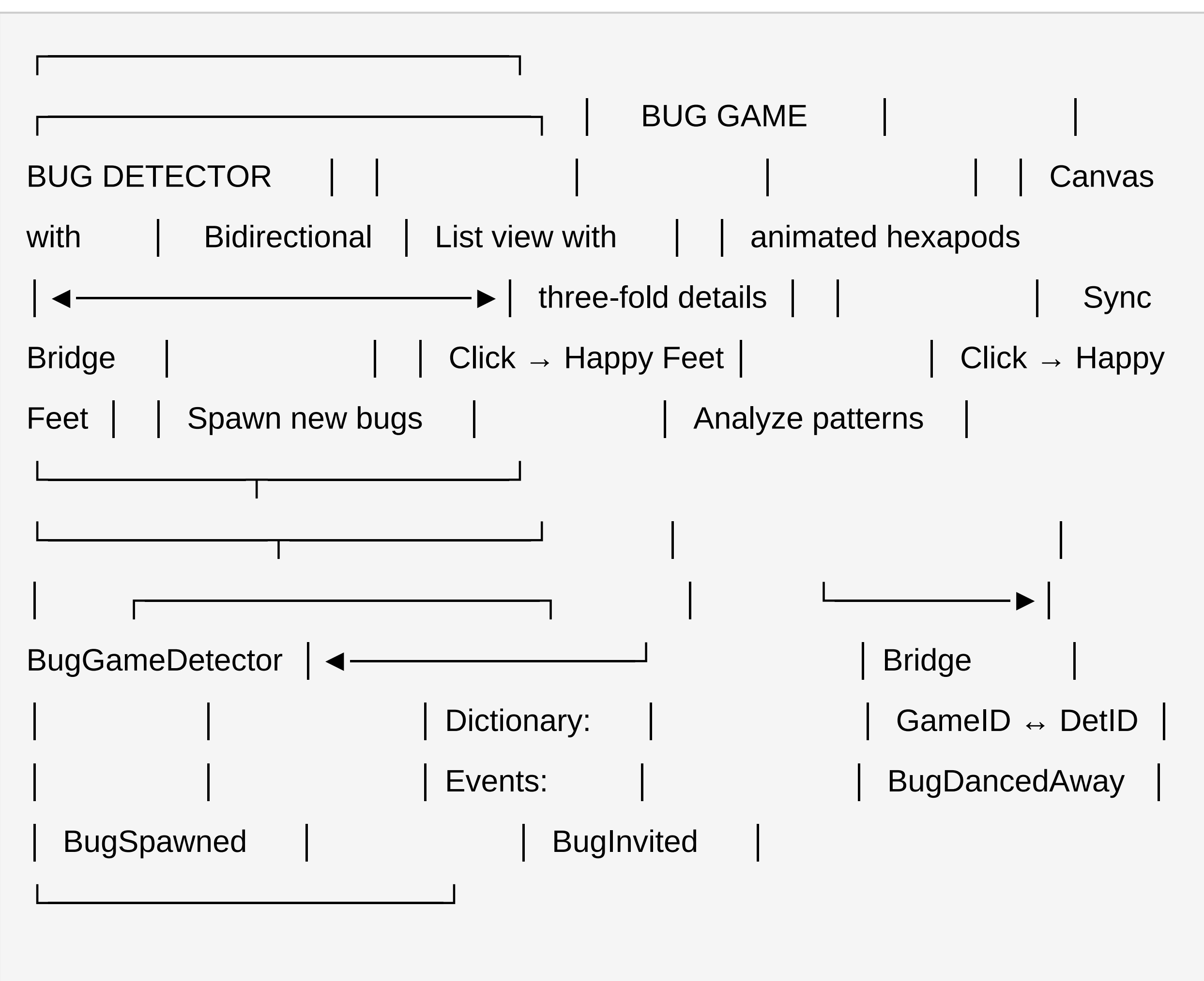
Language Contrast

❌ Traditional (NEVER Use)	✅ QuantumCube (ALWAYS Use)
"Kill the bug"	"Give the bug happy feet"
"Bug terminated"	"Bug danced away"
"Squish bug"	"Invite bug to dance"
"Destroy all bugs"	"Give all bugs happy feet"
"Bug eliminated"	"Bug departed peacefully"
"No bugs remaining"	"All clear — every bug danced away happy!"

8. Bug Game ↔ Bug Detector Pipeline

8.1 Bidirectional Architecture

The Bug Game and Bug Detector are two distinct views of the same underlying bug population. They are connected by a bidirectional synchronization bridge that ensures every action in one view is immediately reflected in the other.



Type Mapping

Each hexapod species in the Bug Game maps to a specific bug type in the Bug Detector:

Game Species	Detector Bug Type	Three-Fold Fault
Fly	H_6 Leak	Zero-iteration loop across CLFL/Machine/HDL
Moth	Phase Desync	Identical flow targets / identical periods
Ant	Holon Collapse	All-zero holon initialization / division by zero
Beetle	Matrix Singularity	Duplicate matrix rows / zero determinant
Mosquito	Freq Drift	Fractional frequency truncation
Cricket	Zone Glitch	Reversed layer ordering / swapped periods

Bidirectional Happy Feet Propagation

When a user clicks a bug in **either** view, the Happy Feet sequence fires in **both** views simultaneously:

6. User clicks Fly in Bug Game canvas
7. Bug Game triggers Happy Feet animation on the clicked Fly
8. Bridge receives `BugDancedAway` event with Game ID
9. Bridge looks up corresponding Detector ID in the dictionary
10. Bridge fires `GiveHappyFeet` on the Detector view with the matched ID
11. Bug Detector removes the H_6 Leak entry with a fade animation
12. Source line cleanup and recompile fire once (not twice)

8.2 BugGameDetectorBridge Design

The bridge maintains a synchronized mapping between the two views:

```
class BugGameDetectorBridge {     // Paired ID mapping: Game Bug ID ↔ Detector Bug ID     Dictionary
```

```
<
Guid, Guid
>
_gameToDetector;    Dictionary
<
Guid, Guid
>
_detectorToGame;    // Core methods:    void SpawnAndRegister(BugType type)
// 1. Create bug in Game view → get gameId        // 2. Create corresponding entry in
Detector → get detectorId        // 3. Store bidirectional mapping     void
OnGameBugDancedAway(Guid gameId)        // 1. Look up detectorId from
_gameToDetector        // 2. Call Detector.GiveHappyFeet(detectorId)        // 3.
Remove mapping entries     void OnDetectorBugDancedAway(Guid detectorId)
// 1. Look up gameId from _detectorToGame        // 2. Call
Game.GiveHappyFeet(gameId)        // 3. Remove mapping entries     // Events
subscribed:    //   Game.BugDancedAway   → OnGameBugDancedAway    //
Detector.BugDancedAway → OnDetectorBugDancedAway }
```

8.3 Marry Views Feature

The Marry Views feature allows both canvases to be superimposed as a single overlay for combined visual inspection.

Aspect	Specification
Trigger	Single-click the 💍 **Marry Views** button
Merge Effect	Both canvases overlay at 50% opacity. A golden ring pulse animation plays on merge (400ms).

Aspect	Specification
Toggle	Click again to **Divorce Views** — canvases separate back to side-by-side layout
Independence	"Cold feet together are not married" — Happy Feet (bug departure) is independent from Marry (view overlay). Two bugs can both get happy feet independently without being married/merged. The views don't need to be married for bidirectional sync to work.

Design Clarification

The Marry Views overlay is purely visual. It does not change the data model or event routing. The BugGameDetectorBridge provides bidirectional sync regardless of whether views are married or divorced. Marry Views is a convenience for visual correlation, not a data operation.

8.4 Game Lifecycle

The Bug Game follows a structured lifecycle with three distinct phases:

Start Phase

- Bugs begin spawning at a configurable rate (default: 1 bug every 2 seconds)
- Each spawned bug is a randomly-selected hexapod species
- The Bridge's `SpawnAndRegister` method creates paired entries in both views
- Canvas displays: **"Game active — bugs incoming!"**

Play Phase

- Bugs walk randomly across the canvas using the alternating tripod gait
- User clicks bugs to trigger Happy Feet departures
- Score tracker displays: **"Bugs Danced Away: X"**
- New bugs continue spawning while the game is active

- Maximum concurrent bugs: 20 (spawning pauses when limit is reached)

Stop Phase

- Spawning ceases immediately
- Remaining bugs freeze in place (legs stop animating)
- All frozen bugs slowly fade to invisible over 3 seconds
- Final message: **"Game complete. X bugs departed peacefully."**
- Score and statistics are preserved for the session

9. C# Companion Architecture (BugsView.cs)

The C# companion code implements the complete QuantumCube Bug Simulator data model, view logic, analysis engine, and bridge architecture. All classes reside in the `QuantumCube.Bugs` namespace.

9.1 Namespace and Enumerations

```
namespace QuantumCube.Bugs
```

BugType Enum

```
enum BugType {     Fly,        // H₆ Leak — winged hexapod     Beetle,     // Matrix Singularity — wingless hexapod     Mosquito,   // Freq Drift — winged hexapod     Moth,       // Phase Desync — winged hexapod     Ant,        // Holon Collapse — wingless hexapod     Cricket     // Zone Glitch — wingless hexapod }
```

BugLayer Enum

```
enum BugLayer {     ClflSource,     // Fold 1: CLFL syntax and grammar     MachineWord,    // Fold 2: ISA-level machine encoding     HdlConstruct    // Fold 3: VHDL entities / Verilog modules }
```

BugState Enum

```
enum BugState {     Active,         // Normal state — walking, visible     HappyFeet,      // Dance + scurry + fade sequence in progress     DancedAway,     // Departure complete, awaiting cleanup     Invisible       // Removed from view, memory eligible for GC     // NOTE: No harm states. No "Dead", "Killed", "Terminated". }
```

SyntaxCategory Enum

```
enum SyntaxCategory {     Keyword,        // LOOP, FLOW, INIT, XFORM, FREQ, etc.     Literal,        // 0, 3.14159, 0.0     Identifier,     // Delta, holon, T1     Operator,       // ->, =     Delimiter,      // [, ], ,     Directive,      // GHz, MHz     Comment         // // text }
```

9.2 Core Classes

ThreeFoldMapping

```
class ThreeFoldMapping {     // Fold 1: CLFL Source     int ClflLineNumber;
```

```
SyntaxCategory ClflCategory;     string ClflSourceLine;          // e.g., "LOOP Delta 0"
string ClflFaultDescription;    // e.g., "Zero iterations"      // Fold 2: Machine Word
uint MachineWord;               // e.g., 0xA0B00000     string MachineOpcode;           //
e.g., "LOOP_INIT"     string MachineFaultDescription; // e.g., "imm=0x0000"      //
Fold 3: HDL Construct     string VhdlConstruct;           // VHDL signal/entity snippet
string VhdlFaultDescription;    // e.g., "counter never increments"     string
VerilogConstruct;        // Verilog reg/module snippet     string VerilogFaultDescription;
// e.g., "tick_count stuck at reset" }
```

BugEntry

```
class BugEntry {     Guid Id;     BugType Type;     BugState State;
ThreeFoldMapping Mapping;      // Anatomy (invariant for all hexapods)     int
LegCount =

>

6;             // Always six     int LegsPerSide =

>

3;          // Always three per side     bool HasWings =

>

// Fly, Mosquito, Moth = true         Type is BugType.Fly or      // Beetle, Ant, Cricket =
false         BugType.Mosquito or         BugType.Moth;      // Visual properties     string
FillColor;               // Species-specific hex color     string StrokeColor;            //
Darker variant of fill     double BodyWidth;              // In pixels     double BodyHeight;
// BodyWidth * aspect ratio }
```

BugView (Mutable Collection)

```
class BugView {     ObservableCollection

<

BugEntry

>

Bugs;      // Mutation methods     BugEntry InviteBug(BugType type);     BugEntry
InviteRandomBug();     void GiveHappyFeet(Guid bugId);     void
GiveAllHappyFeet();      // Events     event Action

<

BugEntry

>

BugInvited;     event Action

<

BugEntry

>

BugDancedAway;     event Action AllClear;         // Fires when last bug departs }
```

BugsProtectedView (Read-Only Wrapper)

```
class BugsProtectedView {     // Read-only access for the Syntax Analyzer     //
Observation only — cannot mutate the collection      IReadOnlyList

<
```

```
BugEntry

>

Bugs;     int Count;     BugEntry GetById(Guid id);     IEnumerable

<

BugEntry

>

GetByType(BugType type);     IEnumerable

<

BugEntry

>

GetByLayer(BugLayer layer);     IEnumerable

<

BugEntry

>

GetByState(BugState state); }
```

9.3 BugSyntaxAnalyzer and Strategies

The `BugSyntaxAnalyzer` operates on the protected (read-only) view and runs randomized analysis strategies to produce insight reports:

```
class BugSyntaxAnalyzer {     BugsProtectedView _protectedView;
AnalysisReport RunAnalysis()     {         // Pick 1–4 random strategies from the pool
of 8         // Execute each selected strategy         // Combine results into a single
```

```
AnalysisReport    }}
```

Eight Analysis Strategies

#	Strategy Name	Description
1	**Syntax Distribution**	Counts bugs per SyntaxCategory (Keyword, Literal, Identifier, etc.) and reports distribution percentages
2	**Machine Word Patterns**	Analyzes the 32-bit machine words for recurring bit patterns: zero immediates, truncated mantissas, duplicate encodings
3	**HDL Fault Correlation**	Cross-references VHDL and Verilog faults to identify bugs that manifest differently in each HDL flavor
4	**Hexapod Wing Correlation**	Compares winged vs. wingless hexapod populations and correlates with fault severity (winged bugs tend to be data-flow faults; wingless tend to be structural faults)
5	**CLFL Grammar Depth**	Analyzes the grammatical complexity of faulted CLFL lines: parameter count, nesting depth, identifier length
6	**VHDL Generic Overlap**	Detects cases where multiple bugs share identical VHDL generic values, indicating systemic configuration issues
7	**Verilog Parameter Drift**	Measures the numerical distance between expected and actual Verilog parameter values across all bugs
8	**Three-Fold Coherence**	Validates that each bug's three-fold mapping is internally consistent: the

#	Strategy Name	Description
		CLFL fault, machine word, and HDL construct all describe the same underlying failure mode

Supporting Classes

```
static class ThreeFoldMappingFactory {     // Creates complete ThreeFoldMapping
instances per bug type     // Randomizes CLFL line numbers for variety     // All
machine words and HDL constructs are deterministic per type     static
ThreeFoldMapping Create(BugType type); }  class BugGameDetectorBridge {     //
Bidirectional sync between Game and Detector views     // Maintains Dictionary

<

Guid, Guid

>

mappings in both directions     // Subscribes to BugDancedAway events on both
views     void SpawnAndRegister(BugType type); }  class MarryViews {     // Single-
click toggle to overlay or separate views     bool IsMarried;     void Toggle();  // Marry
↔ Divorce     // "Cold feet together are not married"     // Happy Feet is independent
from Marry state }
```

9.4 Design Principles

- **Immutable-where-possible:** `ThreeFoldMapping` and `BugsProtectedView` are read-only. Only `BugView` can mutate the bug collection.
- **Event-driven:** All state changes (`BugInvited`, `BugDancedAway`, `AllClear`) are communicated via events, not polling.

- **No harm vocabulary:** No method, property, event, or string literal contains destructive language. The state machine is `Active → HappyFeet → DancedAway → Invisible`.
- **Hexapod-first anatomy:** Every `BugEntry` has `LegCount=6` and `LegsPerSide=3` as computed invariants. There is no setter and no way to create a non-hexapod bug.
- **Three-fold completeness:** No bug exists without a complete mapping across all three layers. The `ThreeFoldMappingFactory` enforces this at creation time.

Appendix A: Quick Reference Card

A.1 Bug Type Quick Reference Table

Type	Species	Legs	Wings	CLFL Fault	Machine Fault	HDL Fault
H_6 Leak	Fly	6	Yes	`LOOP Delta 0` — zero iterations	`0xA0B00000` — imm=0x0000	Counter signal never increments; tick stuck at reset
Phase Desync	Moth	6	Yes	`FLOW Alpha->Alpha->Alpha` — identical targets	`0x41601004` — same period	All generics/parameters identical; no phase offset
Holon Collapse	Ant	6	No	`INIT holon [0,0,0,0]` — all-zero	`0x1100..00` ×4 — imm=0	Magnitude=0; division by zero in normalize
Matrix Singularity	Beetle	6	No	`XFORM holon T1` — duplicate rows	`0x810C/811C` — identical outputs	det_reg/det_out=0; matrix non-invertible
Freq	Mosquito	6	Yes	`FREQ`	`0x2040000A`	Integer

Type	Species	Legs	Wings	CLFL Fault	Machine Fault	HDL Fault
Drift				`3.14159 GHz` — fractional	— mantissa truncated	generic/param truncates to 3
Zone Glitch	Cricket	6	No	`FLOW Delta->Theta->Alpha` — reversed	`0x41600001` — wrong slot	Layer periods inverted; Delta gets Beta's period

A.2 Happy Feet Timing Cheat Sheet

Phase	Duration	Time Window	What Happens
🎶 **Dance**	600ms	0 – 600ms	Legs wiggle (80ms intervals), scale pulse, smile eyes
🏃 **Scurry**	300ms	600 – 900ms	Move to nearest edge at 200px/s, legs continue wiggle
✨ **Fade**	200ms	900 – 1100ms	Opacity 1.0→0.0, stardust sparkle (4–5 dots)
🧹 **Cleanup**	600ms	1100 – 1700ms	Source fade, entity removal, recompile, toast
∑ **Total**	**1,700ms** from single click to complete cleanup		

A.3 Button Label Reference

Label	Action	Notes
🐛 **Invite Bug**	Spawn specific type	Opens species picker; creates paired Game + Detector entries
🦶 **Happy Feet**	Depart selected bug	Also triggered by single-clicking any bug on canvas
🎲 **Random Bug**	Spawn random type	Equal probability across all

Label	Action	Notes
		six hexapod species
🔍 **Detect Bugs**	Run analysis	Scans CLFL source; spawns bugs for detected faults
💍 **Marry Views**	Toggle overlay	50% opacity merge; golden ring animation; click again to divorce
▶ **Start Game**	Begin spawning	Default: 1 bug / 2 seconds; max 20 concurrent
⏹️ **Stop Game**	End game	Freeze remaining bugs; 3-second fade; show final score

---

— End of Document —

QuantumCube Bug Simulator — System Design Document

Version Alpha 0.0001

Prepared by Chas — Auburn, Alabama — April 2026

"Every bug has six legs. No bugs are harmed. One click, happy feet."