Nuts and Bolts: Modular AI from the Ground Up

Kevin Dill
Christopher Dragert
Troy Humphreys
What is Modular AI?

- It’s a way to structure your **AI Architecture**
  - Applies to state machines, behavior trees, HTNs, etc.
- Emphasises small, easily **reused** modules
- Can be **transformative** to your development process
  - Fast prototyping, rapid iteration, increased stability
The Nuts and Bolts

1. Academic Underpinnings (Chris Dragert)
2. Implementation Details with Code Samples (Kevin Dill)
3. Shipped Example and Architecture Discussion (Troy Humphreys)
Nuts and Bolts: Modular AI from the Ground Up

Christopher Dragert, Ph.D.
Ubisoft Toronto
Modular AI

- Software engineering has a lot to say about **modular reuse**
- Apply these principles to **modular AI**
Our Goals

- Learn techniques to develop a suitable modularization for your project
- Understand how to manage and reduce modular complexity
Classifying Complexity

- **Essential complexity**
  - Complexity of the problem itself

- **Accidental complexity**
  - Problems created by us

[Fred Brooks, “No Silver Bullet”, 1986]
What drives Modular Complexity?

1. The **Module** itself
2. Complexity of the **Interface**
3. The **Integration** process
Module Complexity

- Good modules do not try to do too much!

- Smaller modules improve comprehension by having singular purpose
Limiting Scope

- Separate *cross-cutting concerns*
  
  *Example* - Melee combat module selects a target, ranged module selects a target, flee module selects a target…

- *Solution* - Remove target selection from existing modules, create a *target selection module*
Control the Size

- Traditional abstraction techniques should be applied
  - Hierarchical Approaches
  - Subsumption and Layering
  - Parallelism
Well-Defined Semantics

- Your AI logic must operate in a understandable, well-defined fashion
- Necessary for portability between games
Semantics Example

- What transition does your implementation take?
  - The new context must make the same choice!
Modular Interface

- Communicates the required context for the module
- Raises the level of abstraction, reducing accidental complexity
Defining the Context

- State machines (event-based formalisms)
  - What **input events** in do you need to handle?
  - What **output events** do you generate?
Enemy Position Tracker

Description: Tracks the position of an enemy.

Game: Ubisoft Open World Game

Parameters: `<T>` The type of the enemy entity

Language: C++

Input Events:
- ev_EnemySpotted(<T> enemy)

Output Events:
- ev_EnemyLost(<T> enemy)
- ev_EnemyPositionChanged(<T> enemy)
**Enemy Position Tracker**

*Description*: Tracks the position of an enemy  
*Game*: ‘Game X’ by Ubisoft  
*Parameters*: `<T>` The type of the enemy entity  
*Language*: C++
## Enemy Position Tracker

**Description**: Tracks the position of an enemy

**Game**: ‘Game X’ by Ubisoft

**Parameters**: `<T>` The type of the enemy entity

**Language**: C++

### Input Events

- ev_EnemySpotted(<T> enemy)
- ev_EnemyLost(<T> enemy)
## Enemy Position Tracker

**Description**: Tracks the position of an enemy

**Game**: ‘Game X’ by Ubisoft

**Parameters**: `<T>` The type of the enemy entity

**Language**: C++

<table>
<thead>
<tr>
<th>Input Events</th>
<th>Output Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>- ev_EnemySpotted(&lt;T&gt; enemy)</td>
<td>-ev_EnemyPositionPositionChanged(&lt;T&gt; enemy)</td>
</tr>
<tr>
<td>- ev_EnemyLost(&lt;T&gt; enemy)</td>
<td></td>
</tr>
</tbody>
</table>
Behavior Tree Contexts

- Primarily data-driven
  - What **blackboard entries** are read (input) and written (output)?

- Not the full story!
Behavior Tree Contexts

- New behavior trees where nodes can return \{success, failure, running\}
  - What \textit{interrupts} a running node?
- Tree structure itself
Behavior Tree Interfaces

Interface Complexity

Check Conditional

∞ → Reuse!

∞ → Reuse?

∞ → ...
Integration Overview

- The essential problem is connecting inputs and outputs between modules

- Everything else is accidental complexity!
Integration Complexity

- Module **connections** must be derivable solely from the interface
  - This preserves **modular encapsulation**!

- A consistent integration approach can be supported with **tools**
Module Coupling

- **Loosely-Coupled**: A missing module impairs only that behavior

- Loosely coupled modules support fast prototyping and rapid iteration
Module Coupling

- **Tightly-Coupled**: Missing modules cause failures, prevent compilation, etc.
  - Often caused by *broken encapsulation*
  - Could also be an error in abstracting modular concerns
Special Cases

- Special case exceptions break reuse
  - **Sensor**: Reports every `ev_newEnemySpotted` event
  - **Reaction**: `ev_newEnemySpotted` causes a new enemy reaction
  - Event system adds hysteresis, caps generation of `ev_newEnemySpotted` at one per minute
- This is a broken module encapsulation error
The Payoff

- **Fast Prototyping**
  - Quickly modify functionality by adding and removing modules

- **Fine Tuning**
  - Parameterized module instances allow for customization

- **Better Development Process**
  - Reuse existing behavior, spend time innovating new behaviors
A good modular approach:

- Uses **small modules** that separate concerns
- Operates with **well-defined semantics**
- Has a **clear interface**
- Preserves modular **encapsulation**
- Uses a **loose-coupling** approach