VIETNAMESE-GERMAN UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Frankfurt University of Applied Sciences Faculty 2: Computer Science and Engineering

Enhancing Modularity and Performance in Real-Time Strategy Games Using Entity-Component-System

Author:	Le Huynh Dong Quan
Program:	Bachelor of Science in Computer Science and Engineering
VGU Student ID:	10421131
First supervisor:	Dr. Le Lam Son
Second supervisor:	Mr. Le Duy Hung
Intake:	2021 - 2025

BACHELOR THESIS
Submitted in partial fulfillment of the requirements for the degree of Bachelor Engineering in study program Computer Science, Vietnamese - German University, 2025

AUGUST 2025

(Intentionally left blank)

Declarations

I, Le Huynh Dong Quan, affirm that this thesis is a product of my own work that I conducted at Vietnamese-German University. This thesis was completed under the supervision of Dr. Le Lam Son and Mr. Le Duy Hung. Furthermore, I declare that all opinions, results, conclusions are my own and may not represent the policies or opinions of the Vietnamese German University

1 Acknowledgments

I would like to thank my thesis supervisors, for their wonderful patience and feedback during this project.

Thanks to my friends and fellow students who helped test and review the prototype, listened to my rants, or just kept me sane throughout the process.

I'm especially grateful for the support and encouragement from my family, without which this work would not have been possible.

And finally, thanks to all the caffeine and late-night debugging sessions that made this thesis possible.

2 Abstract

This thesis presents a comparative study between traditional Object-Oriented Programming (OOP) and the Entity Component System (ECS) architecture within the context of game development using Unity. A real-time prototype was developed in two parallel implementations, one using Unity's MonoBehaviour-based OOP model, and the other using Unity DOTS (Data-Oriented Technology Stack) with ECS. The study aims to evaluate and contrast the two paradigms across several dimensions, including performance, code complexity, scalability, parallelism, and maintainability.

Quantitative metrics such as main thread time, frame rate, and garbage collection behavior were collected through automated benchmarking of simulated agent behavior under increasing unit counts. Qualitative aspects such as code modularity, boilerplate, and extensibility were analyzed through practical case studies and design examples drawn from the prototype.

Results demonstrate that ECS provides superior runtime performance and improved parallelism support, particularly under high-entity simulations. However, this comes at the cost of increased boilerplate, steeper learning curves, and greater architectural rigidity. The findings aim to inform developers of the trade-offs between these approaches, helping guide architectural decisions in performance-critical or scalable game systems.

Results demonstrate that ECS provides superior runtime performance and improved support for parallelism, particularly under high-entity simulations. In contrast to the inheritance-based structure of OOP, the ECS approach encourages decoupled and modular system design, which improves long-term maintainability and feature isolation. However, these benefits come at the cost of increased boilerplate, a steeper learning curve, and reduced intuitiveness due to its data-oriented design. The findings aim to inform developers of the practical trade-offs between these approaches, supporting architectural decisions in performance-critical or scalable game systems.

Acronyms

Acronym	Definition
OOP	Object-Oriented Programming
ECS	Entity Component System
DOTS	Data-Oriented Technology Stack
RTS	Real-Time Strategy
ID	Identity
UI	User Interface
FPS	Frames Per Second
GC	Garbage Collection

List of Figures

List of Tables

List of Code Snippets

Contents

3 Introduction

In modern game development, software architecture plays a critical role in shaping the scalability, performance, and maintainability of a project. As games grow increasingly complex—with hundreds or even thousands of active entities—traditional paradigms such as Object-Oriented Programming (OOP) often expose limitations, including performance bottlenecks, tight coupling between systems, and reduced flexibility in adapting to evolving gameplay requirements.

In response to these challenges, data-oriented design has gained popularity, particularly through the adoption of the Entity Component System (ECS) architecture. ECS shifts focus from objects and inheritance to the organization of data and behavior in a way that maximizes performance and parallel execution.

This thesis investigates the practical differences between traditional OOP and ECS within the context of Unity, by developing a RTS prototype using both paradigms. The prototype features autonomous agent units operating in a real-time environment, implemented in two separate versions: one using MonoBehaviour-based OOP, and the other using Unity's DOTS-based ECS framework.

The study evaluates both approaches through quantitative and qualitative analysis. Performance is measured in terms of CPU usage, frame rate, memory allocation, and garbage collection, while code structure is compared through maintainability, modularity, and extensibility. Additional criteria such as parallelism support, boilerplate code, and developer ergonomics are also considered.

This research aims to provide a balanced, practical evaluation of the ECS architecture in comparison to traditional OOP, supporting developers in making informed architectural choices for game development.

4 Background and Related Work

This section presents an overview of the relevant background information, focusing on the two target game architectures: Object-Oriented Programming (OOP) and Entity-Component-System (ECS). It also discusses the technologies used in the development of the two versions of the game, each utilizing one of these architectures.

4.1 RTS Games

In an RTS game, players control units and structures in real time from an overhead perspective. The goal is to gather resources, build an army, and defeat the opponent by destroying their forces. Unlike turn-based games, RTS players can perform as many actions as they can manage while the game runs continuously. Notable RTS games include Dune II, Total Annihilation, and the Warcraft, Command & Conquer, Age of Empires, and StarCraft series.

Each match starts with players controlling a few units and structures on a 2D map, with resources scattered around for gathering. Players must expand and manage their economy, infrastructure, and military, balancing between these aspects to maintain an effective army. Strategic points on the map require attention for attack and defense. Matches end when a player destroys all the opponent's units and structures, though players often concede early if defeat seems inevitable. [1]

RTS games requires efficient management of various entities (e.g., units, buildings, resources) and systems (e.g., AI, pathfinding, combat mechanics) to ensure smooth gameplay, especially when the game state grows in complexity over time. This presents challenges in game design and software architecture, which motivates the comparison of different programming paradigms like OOP and ECS.

Figure 1: Age of Empires - A classic example of RTS games

4.2 Common Game Architectures

Two common architectures used in game development are Object-Oriented Programming (OOP) and Entity-Component-System (ECS). Both approaches have advantages and trade-offs, particularly when developing performance-intensive applications like RTS games.

4.2.1 Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a widely-used paradigm where the game world is modeled as a collection of objects that interact with each other. Each object is typically an instance of a class and encapsulates both data and behaviors. In the context of an RTS game, entities like units, buildings, and resources are often represented as objects with methods that define their actions (e.g., move, attack, extract resources).

While OOP allows for clear hierarchies and reusability through inheritance and polymorphism, it can face performance bottlenecks in scenarios where a large number of entities must be processed in real time, such as in RTS games. The overhead associated with object management, such as dynamic memory allocation and garbage collection, can impact the efficiency of

an OOP-based system, especially in the case of games with a large number of active objects.

4.2.2 Entity-Component-System (ECS)

The Entity-Component-System (ECS) is an architectural pattern that focuses on composition over inheritance. In ECS, entities are simple identifiers (usually just IDs) that are composed of various components. Components are data containers, and systems operate on entities with specific components, processing them in batches to optimize performance.

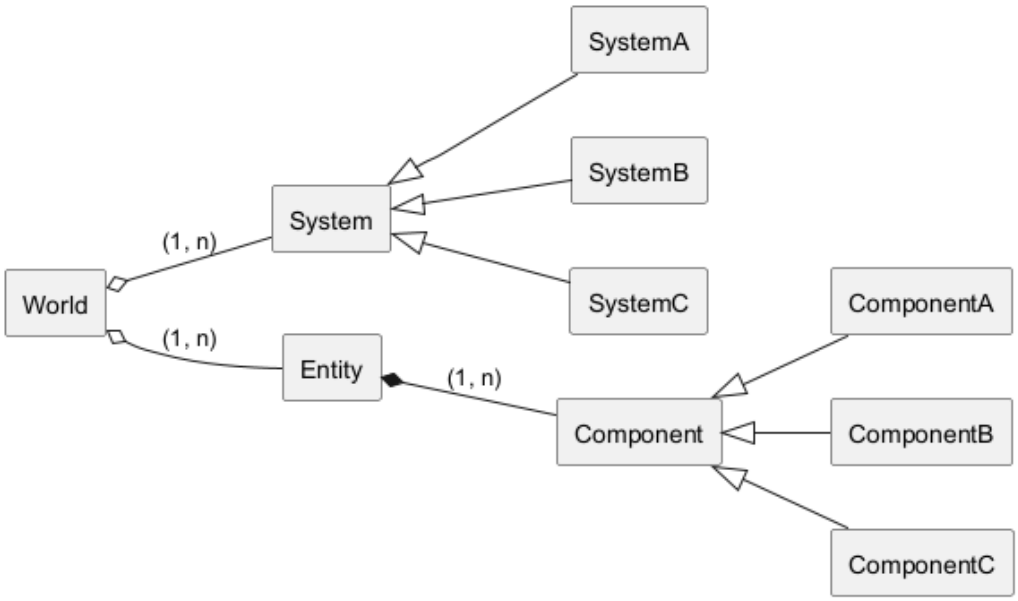


Figure 2: ECS Concept Diagram
The World contains Entities and Systems. Each Entity is composed of one or more Components. Each System process on the relevant Components. [5]

ECS is particularly well-suited for games with many entities, such as RTS games, because it allows for better memory layout and cache-friendly processing. By separating data from behavior and processing entities in bulk (often using multithreading), ECS can improve both performance and scalability. Unity's Data-Oriented Technology Stack (DOTS) is a prominent example of ECS applied in game development, offering high performance for large-scale simulations.

However, while ECS offers superior performance and flexibility, it requires a different mindset than OOP. Developers must design systems and components independently, and managing

the relationships between entities can be more complex than in OOP, especially for games with more intricate mechanics.

4.3 Relevant Technologies

The game versions discussed in this paper were developed using Unity, a widely used game engine that supports both OOP and ECS architectures. The following sections describe the relevant technologies and frameworks employed in each version.

4.3.1 Unity and GameObjects (OOP Version)

Unity is a powerful and flexible game engine used for the development of both 2D and 3D games. In the OOP version of the game, Unity's GameObject-based system was utilized. Each unit, building, and other interactive elements in the game are represented as GameObjects, which can be extended with components such as `Transform`, `Collider`, and `Script` components that define the object's behavior[2].

Unity's traditional GameObject architecture is straightforward and benefits from Unity's vast ecosystem, making it easy to implement complex game mechanics quickly. However, this approach can become inefficient in resource-intensive scenarios due to the overhead of Unity's object management system[3].

4.3.2 Unity DOTS (ECS Version)

Unity's Data-Oriented Technology Stack (DOTS) is an advanced framework designed to utilize the Entity-Component-System (ECS) paradigm for high-performance game development. DOTS allows developers to write code that runs efficiently on modern multi-core processors by providing fine-grained control over memory layout and parallel execution.

The ECS version of the game was developed using Unity DOTS, which handles entities and components in a way that minimizes memory overhead and optimizes cache utilization. DOTS also supports multithreading and burst compilation, which can significantly improve the performance of RTS games that require the processing of many entities and real-time simulation[4]

5 Implementation Details

5.1 Prototype Overview

The prototype simulates core RTS gameplay features including pathfinding, combat interactions, team logic, and resource gathering. It is implemented in two versions: a MonoBehaviour-based model using traditional OOP, and a version built on Unity's DOTS employing the ECS architecture. Both versions share a consistent feature set, ensuring a fair comparison of their architectural differences.

5.1.1 Prototype Gameplay

The prototype begins by allowing the player to choose from a set of predefined maps or generate a custom map. Each map contains information about all the entities present in the game, including their positions, types, and team affiliations, offering flexibility for testing various game scenarios. When the simulation starts, the prototype automatically populates the selected map with units, structures, and resources, emulating the initialization phase of a traditional RTS game. Once initialized, entities behave according to their assigned logic, performing actions such as movement, combat, and resource extraction. This setup enables focused testing of individual and group behaviors in different scenarios. The match progresses until only one team remains with surviving units, implementing a basic win condition that captures the core competitive nature of RTS gameplay.

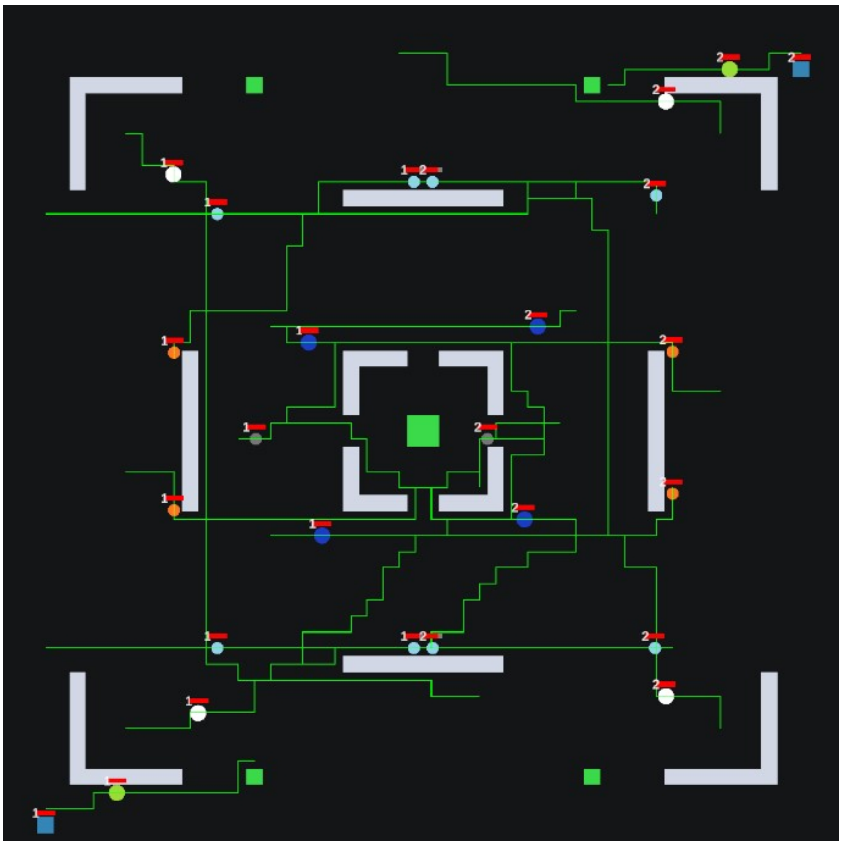


Figure 3: The prototype in play mode featuring all the unit types

5.1.2 Core Mechanics

The prototype operates based on a set of core mechanics that define the overarching game logic and govern the interactions among all entities. The game is set within a 2D grid-based environment, where each entity occupies a tile and navigates the map accordingly. Pathfinding is implemented using the A* algorithm, allowing units to efficiently traverse the grid while avoiding obstacles.

Each unit is equipped with a health value and can engage in combat, enabling the exchange of damage during encounters. The logic is team-based, with units and structures assigned to specific teams, which influences their interactions and targets. At the beginning of each match, entities are initialized according to a predefined map layout.

No new units are spawned during runtime, ensuring that the game evolves solely based on the initial conditions and player decisions.

5.1.3 Entities

The prototype defines a variety of unit types, each with distinct behaviors and responsibilities:

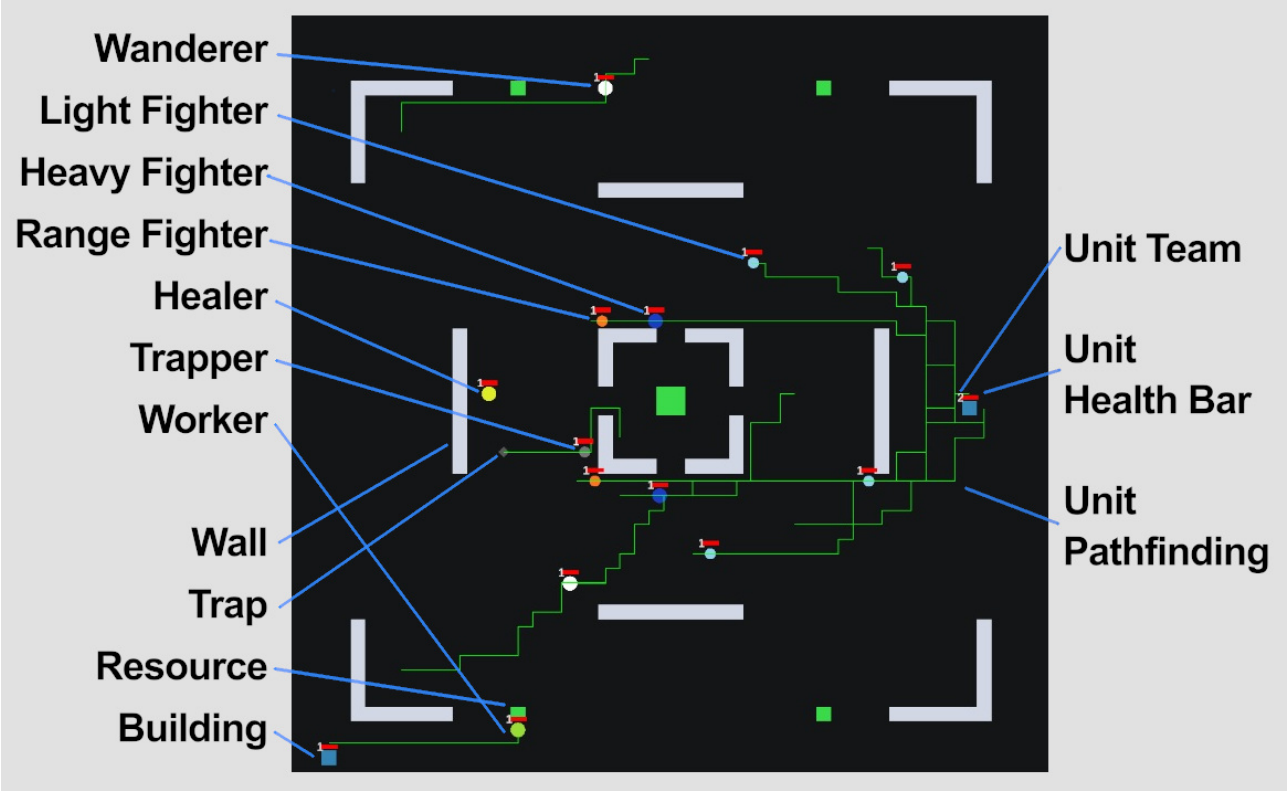


Figure 4: The prototype in play mode featuring all the unit types

Table 1: Unit Types and Their Roles

Unit Type	**Description**
Wanderer	Moves randomly throughout the environment, simulating exploratory behavior.
Fighter	Engages enemies in combat. Includes three subtypes: *Light*: Fast, average health. *Heavy*: Slow, high health. *Range*: Attacks from a distance, low health.
Healer	Restores the health of nearby allied units within a specific range.
Worker	Collects resources and constructs buildings for the team.
Trapper	Deploys traps that damage enemy units.

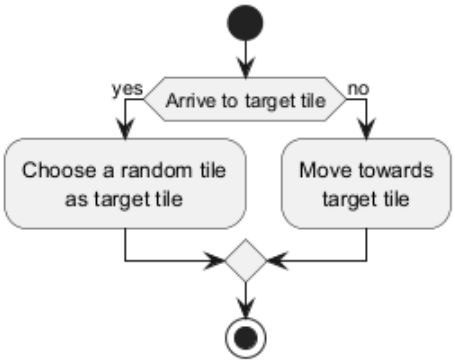


Figure 5: Wanderer flowchart

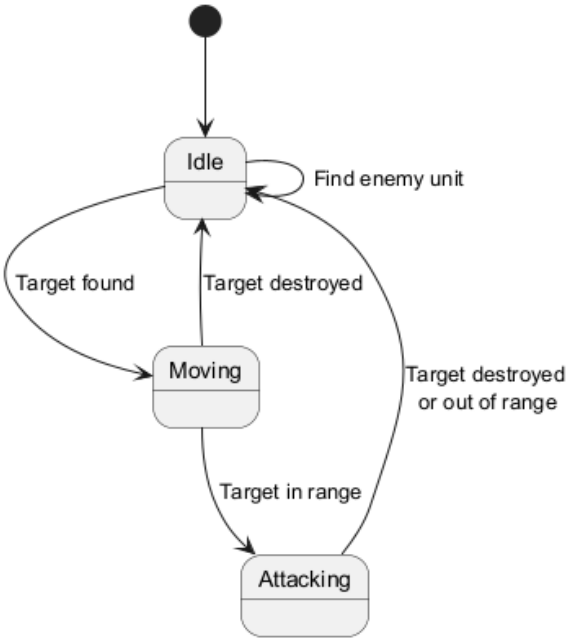


Figure 6: Fighter state diagram

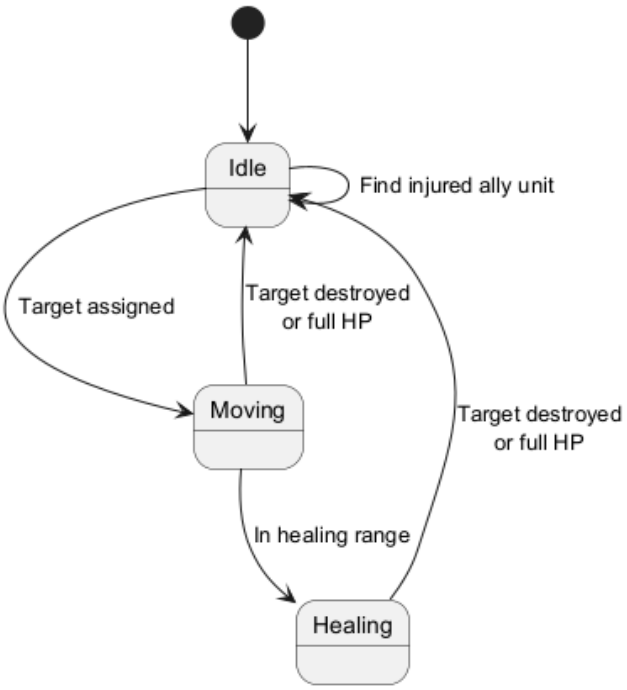


Figure 7: Healer state diagram

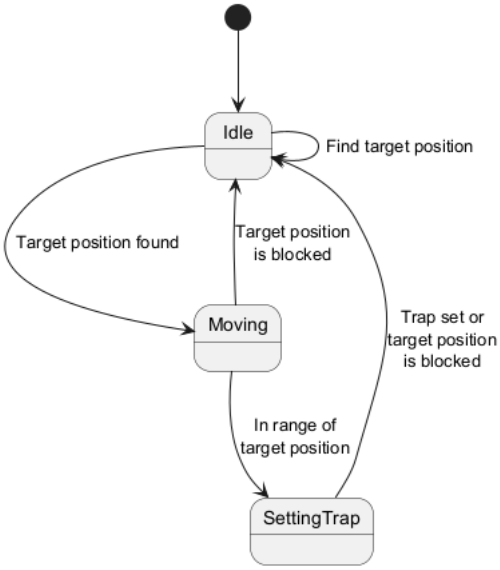


Figure 8: Trapper state diagram

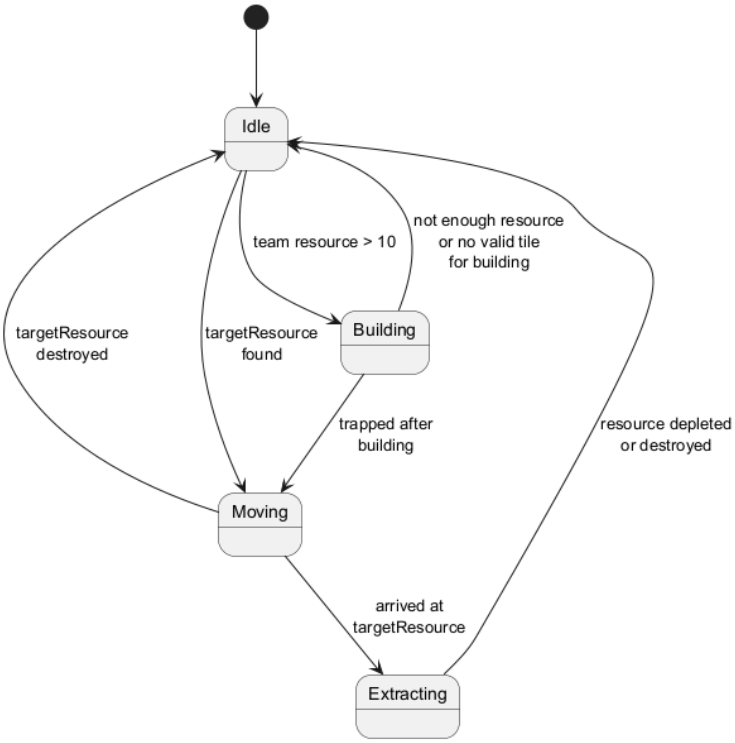


Figure 9: Worker state diagram

In addition to player-controlled units, the prototype includes various static entities that contribute to gameplay mechanics:

Table 2: Static Entites

Static Entity	**Description**
Walls	Indestructible, neutral barriers that obstruct unit movement and shape the map layout.
Resources	Harvestable nodes collected by Worker units to enable building construction.
Buildings	Structures constructed by Workers. They also block movement and can be attacked and destroyed by enemy units.
Traps	Deployed by Trapper units. These entities automatically damage enemy units upon entering the occupied tile.

5.2 OOP-based Implementation

The OOP version of the RTS prototype is implemented using Unity's traditional `MonoBehaviour` system. In this architecture, each game entity is attached `GameObject` with a custom script that encapsulate state and behavior. These scripts are implemented as C# classes derived from `MonoBehaviour`, Unity's base class for custom game logic.

5.2.1 Entity Representation

Each unit is represented as a `GameObject` with associated scripts that encapsulate its behavior. For instance, a *Healer* is assigned the `Healer.cs` script, which governs its healing logic, while a *Light Fighter* uses `Fighter.cs` with specific parameter configurations. Static entities such as buildings, resources, traps, and walls are also implemented as `GameObjects`, each paired with behavior-specific scripts that govern their interactions and properties.

5.2.2 Control and Coordination

Game-wide coordination is handled by centralized manager classes. The `GameManager` oversees initial entity spawning and game state progression. The `ResourceManager` tracks resource accumulation for each team. Units interactions and gameplay logic is distributed across individual `Update()` methods.

5.2.3 Movement and Pathfinding

All units use a classic A* pathfinding system implemented in the `AStarPathfinder.cs` script and referenced in the `Unit.cs` script. This algorithm calculates paths across a 2D grid and returns a series of waypoints for the unit to follow.

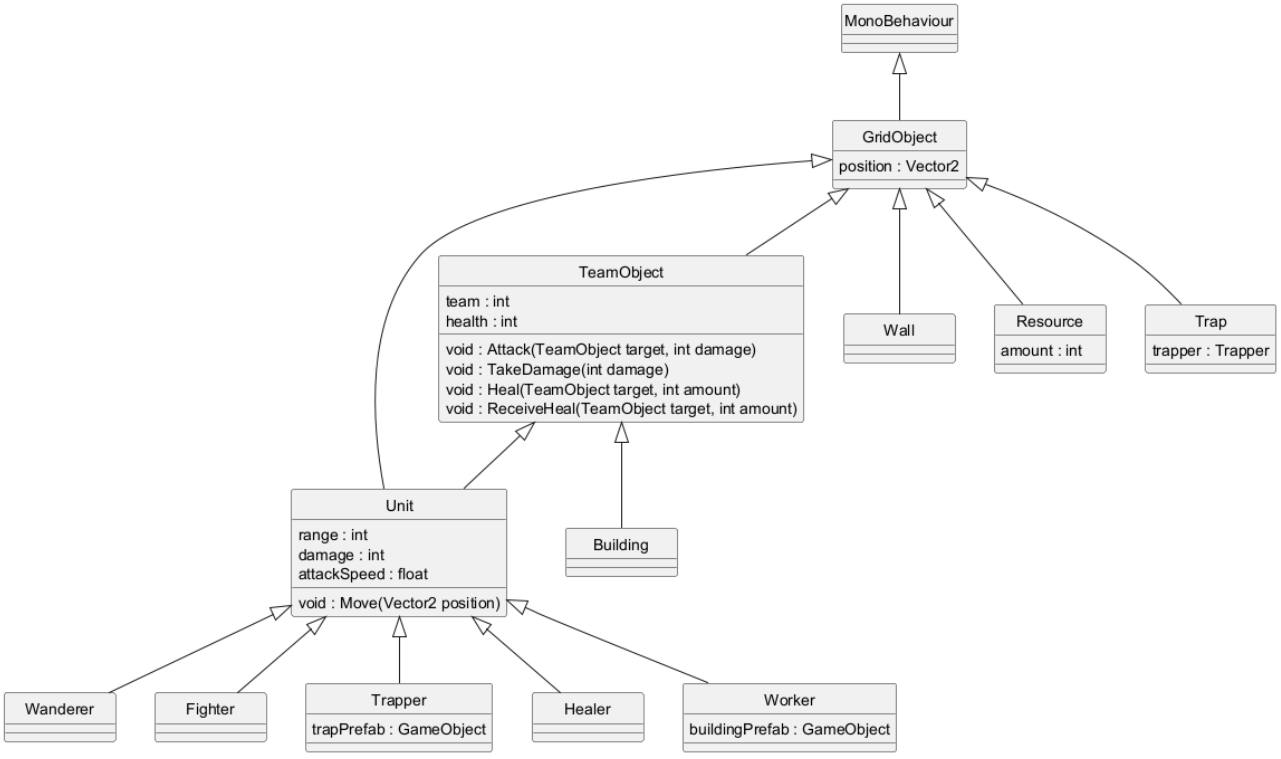


Figure 10: OOP Implementation class diagram

Each leaf class is a script that defines the behavior of a specific entity and is attached to the corresponding GameObject.

5.3 ECS-Based Implementation

The ECS version of the RTS prototype is designed using Unity's Entity Component System (ECS), leveraging the DOTS (Data-Oriented Technology Stack) framework. Unlike the OOP approach, ECS emphasizes data-driven architecture, where entities are pure data containers, components are raw data structs, and systems operate on groups of components in a performant and multithreaded fashion.

5.3.1 Entity Representation

Entities are represented as lightweight identifiers managed by Unity's `EntityManager`. Instead of behavior-heavy scripts, all logic is driven by systems that act on components. Units, buildings, and resources are defined through combinations of pure data components such as `HealthComponent.cs`, `UnitComponent.cs`, or `ResourceComponent.cs`. There are no per-entity update methods, and all interactions are system-driven.

5.3.2 Components

Each entity's behavior is determined by the set of components it possesses. For example, a unit with `GridPositionComponent.cs`, `TeamComponent.cs`, `HealthComponent.cs`, `UnitComponent.cs`, and `HealerComponent.cs` is recognized as a Healer unit. Components are implemented as `IComponentData` structs to enable cache-friendly memory access and parallel processing.

5.3.3 Systems

All game logic is executed by systems that operate on entity groups matching specific component components. Systems such as `UnitSystem`, `FighterSystem`, and `WorkerSystem` process all relevant entities in batches. This architecture enables high performance through multithreading and SIMD-friendly operations.

5.3.4 Pathfinding and Movement

Pathfinding is handled by an ECS-compatible A* algorithm implemented in the `ECSAStarPathfinder.cs` script, which computes paths across a 2D grid. Similar to the OOP

structure, the `UnitSystem.cs` references this script to request a new path when a flag in the `UnitComponent.cs` is triggered.

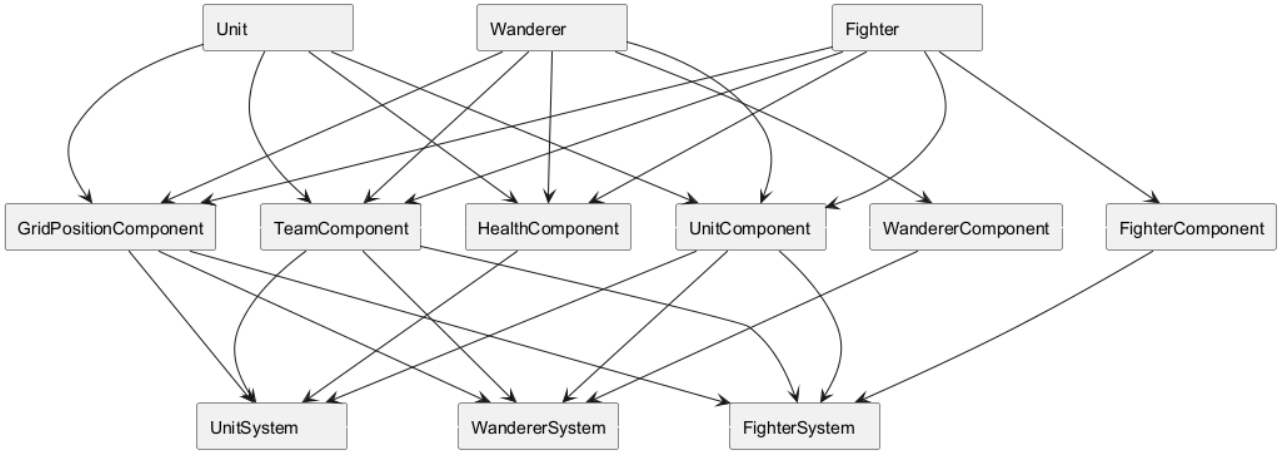


Figure 11: ECS Implementation diagram 1

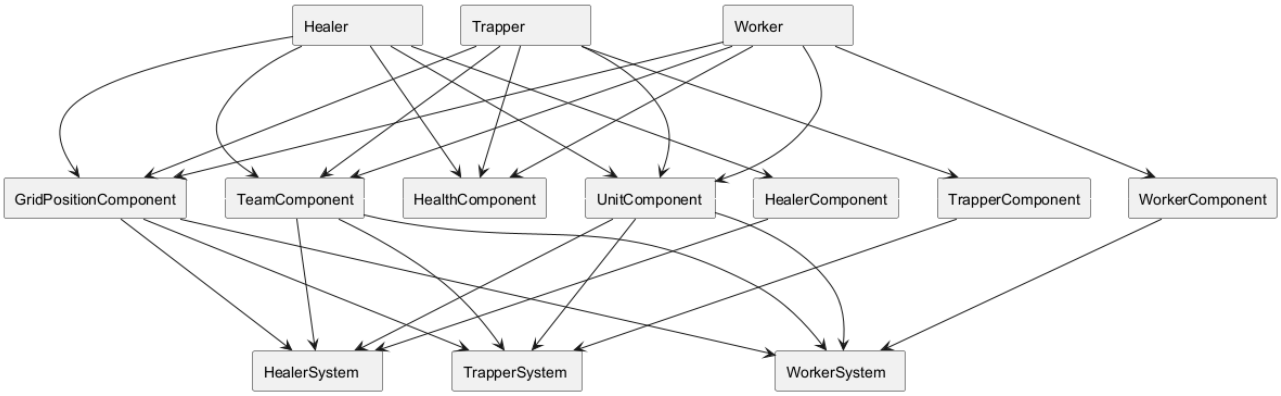


Figure 12: ECS Implementation diagram 2

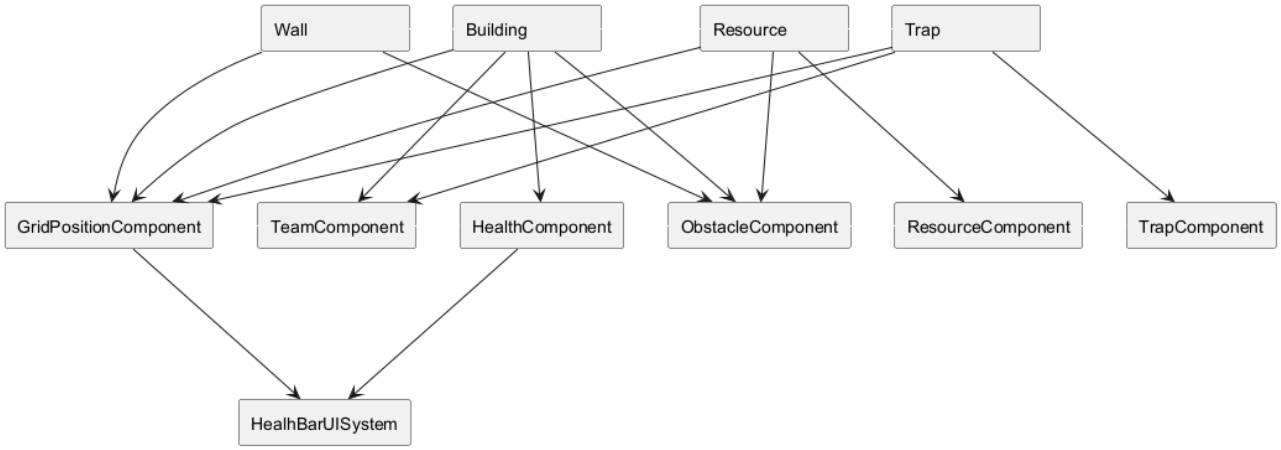


Figure 13: ECS Implementation diagram 3

6 Evaluation and Comparison

6.1 Performance

A key performance advantage of ECS lies in its cache-friendly memory layout. Traditional OOP structures often scatter object data across memory, leading to frequent cache misses during iteration. In contrast, ECS organizes data in contiguous arrays of components, allowing systems to process large batches of entities linearly in memory. This improves CPU cache utilization, reduces memory access latency, and enables more efficient parallelization. As a result, ECS architectures can achieve significantly better performance in large-scale simulations compared to typical OOP designs. [6] [7]

6.1.1 Test setup

To evaluate the scalability and performance characteristics of both implementations, three separate test scenarios were conducted. Each test used a different number of Wanderer agents deployed in the Ruins map: 250, 350, and 500 units, respectively. These variations were chosen to simulate increasing computational load and to observe how each architecture responds to larger agent populations under identical environmental conditions.

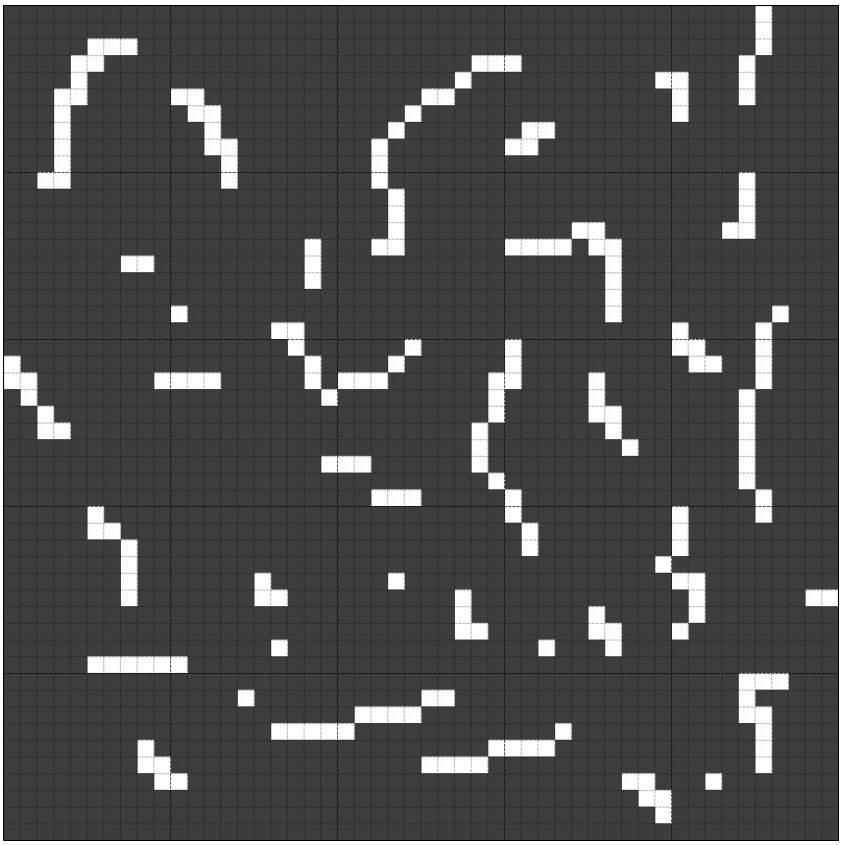


Figure 14: A 50x50 2D map titled "Ruins"

6.1.2 Comparison Methodology

Following the establishment of the test environment, the next phase of this study involves a systematic comparison of performance characteristics between the Object-Oriented Programming (OOP) and Entity Component System (ECS) implementations of the Wanderer agents. To ensure a meaningful and reproducible comparison, a set of quantitative metrics was selected to evaluate each architecture's efficiency, scalability, and runtime behavior.

The comparison focuses primarily on three key performance indicators:

CPU Usage (Main Thread Time): Measures the average time spent per frame on processing logic, particularly in the update cycle. This is crucial for understanding the execution overhead of each architecture and identifying any potential bottlenecks'

Frame Rate (Frames Per Second): Captures the real-time responsiveness of the system by monitoring how consistently the simulation maintains high frame rates. Frame rate instability may indicate inefficiencies or performance spikes within the underlying architecture'

Memory Usage and Garbage Collection: Observes the volume of heap allocations and frequency of garbage collection events. OOP implementations often result in frequent allocations and managed memory churn, whereas ECS tends to reduce GC pressure through improved data locality and static memory layouts'

All measurements were conducted using Unity's built-in Profiler. Each test was repeated multiple times for a period of 5 minutes under identical conditions to ensure consistency and long-term performance, and averages were computed to reduce the effect of transient anomalies. This methodical approach enables a direct, empirical comparison between the traditional OOP structure and the data-oriented ECS paradigm, providing the foundation for the results and discussion presented in the subsequent section'

Table 3: Performance Comparison Between OOP and ECS Implementations

Configuration	Avg. Main Thread Time (ms)	Avg. Frame Rate (FPS)	Allocated Memory (KB/frame)	GC Frequency (per min)
OOP (250)	23.08	45	495.86	13
ECS (250)	16.64	96	9.09	1
OOP (350)	67.37	18	1045.23	11
ECS (350)	16.77	58	14.7	1
OOP (500)	309.43	3	4435.43	8
ECS (500)	25.25	40	17.12	1

6.2 Code Structure and Maintainability

6.2.1 Side Effects

Side effects are observable changes in state that occur when a function or expression is evaluated — beyond just returning a value[5]. In traditional object-oriented game architecture, function calls often embed not only logical operations but also direct state changes or actions with unintended consequences. This can lead to hidden dependencies and reduced maintainability.

Example: Side Effects in a Unit Attack

```
// TeamObject.cs

protected void Attack(Unit target)
{
  target.TakeDamage(damage);
}

public void TakeDamage(int damage)
{
  health -= damage;
  if (health <= 0)
  {
      health = 0;
      Destroy(gameObject); // Side effect: immediate destruction
  }
  if (healthSlider != null)
      healthSlider.value = health;
}
```

Code snippet 1: OOP unit attack with embedded side effect

At first glance, the `Attack()` method appears straightforward. However, calling `TakeDamage()` triggers a *side effect* — the potential destruction of the target GameObject. This introduces **non-obvious control flow**: the attacker indirectly affects the game world beyond the damage calculation.

Such implicit behavior can complicate debugging and testing. For example, if another system assumes the target still exists after `Attack()`, it may encounter null references or inconsistent state.

A more maintainable approach involves **separating calculation from execution**, isolating side effects into clearly defined phases (e.g., damage resolution vs. entity destruction). This principle is more naturally supported in ECS architectures, where systems often queue structural changes like entity removal for the end of a frame.

```
// FighterSystem.cs

ecb = new EntityCommandBuffer(Allocator.TempJob);

```

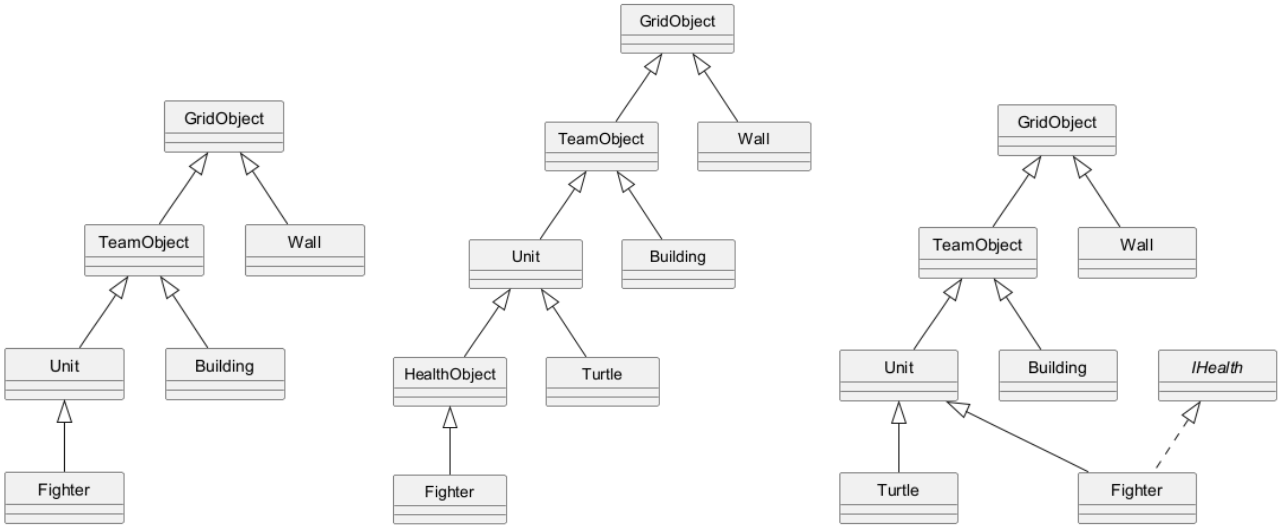
```
// Query loop
RefRW<HealthComponent> hc = SystemAPI.GetComponentRW<HealthComponent>(
    fighterComponent.ValueRW.target);
hc.ValueRW.health -= unitComponent.ValueRO.damage;
if (hc.ValueRW.health <= 0)
{
  ecb.DestroyEntity(fighterComponent.ValueRW.target); // Queue the
    destruction instead of destroying the entity immediately
}

ecb.Playback(state.EntityManager);
ecb.Dispose();
```

Code snippet 2: ECS unit attack with centralized side effect

6.2.2 Composition over Inheritance

Developing a game system is challenging due to the unpredictable nature of future feature requests. Consider a scenario where a new unit, Turtle, is introduced. This unit is unique in that it cannot be damaged (and thus does not require a health system), but it still needs to move and be assigned to a team—just like other units.

In the OOP implementation, one workaround is to introduce a new class called HealthObject, placed as a subclass of Unit, and move all health-related logic from TeamObject into HealthObject. The existing unit classes would then inherit from HealthObject, while Turtle would inherit directly from Unit to avoid the health functionality. Another alternative is to extract the health logic into an IHealth interface and selectively implement it in the appropriate subclasses. However, both solutions require restructuring the class hierarchy or altering existing code, which violates the Open/Closed Principle of SOLID design — that software entities should be open for extension but closed for modification.



(a) Original Structure (b) Introduce New Class (c) Introduce New Interface

Figure 15: OOP Feature Addition

In contrast, the ECS implementation handles this scenario with minimal changes. Since unit data is represented as a composition of components, such as GridPositionComponent, UnitComponent, TeamComponent, and HealthComponent, the Turtle entity can be created without a HealthComponent. The absence of the component is sufficient to exclude it from health-related systems, making the architecture naturally extensible without altering existing code.

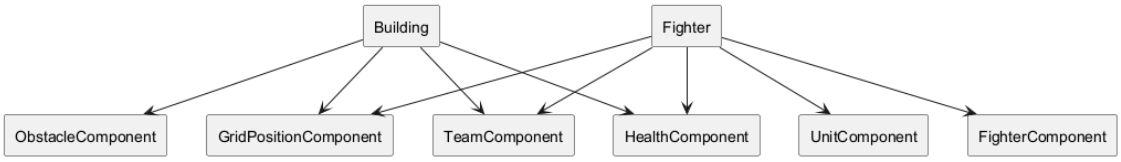


Figure 16: ECS Original Structure

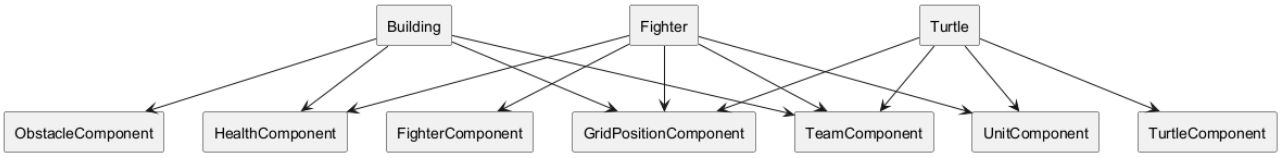


Figure 17: New Entity Addition

6.2.3 Decoupling

In the prototype, the health value plays multiple roles: it is modified during combat, used to determine if a unit is dead or fully healed, and displayed through the health bar. In the traditional OOP implementation, all of these behaviors are handled within the `TeamObject` class, resulting in tight coupling between unrelated concerns.

```
public class TeamObject : GridObject {
  public int health;
  public int maxHealth;
  private Slider healthSlider;

  public void TakeDamage(int damage)
  {
    health -= damage;
    if (health <= 0)
    {
      health = 0;
      Destroy(gameObject);
    }

    healthSlider.value = health;
  }

  public void ReceiveHeal(int heal)
  {
    health = Mathf.Max(health + heal, maxHealth);

    healthSlider.value = health;
  }

  public bool IsFullHealth()
  {
    return health >= maxHealth;
  }
}
```

Code snippet 3: Health means different things to different systems at different times)

In contrast, the ECS implementation treats the `HealthComponent` as a purely data-holding

structure, containing only the current health value. Each behavior is instead managed by its own dedicated system such as `FighterSystem`, `HealerSystem`, and `HealthBarUISystem`. This separation allows the same health value to serve different purposes across systems, depending on the context and timing of execution.

```
public struct HealthComponent : IComponentData {
    public int health;
    public int maxHealth;
}

// In FighterSystem
healthComponent.ValueRW.health -= damage;
if (healthComponent.ValueRW.health <= 0)
{
  ecb.DestroyEntity(entity);
}

// In HealerSystem
healthComponent.ValueRW.health = math.min(entityHealth.Value + healAmount,
   healthComponent.ValueRW.maxHealth);

// In HealthBarUISystem
healthSlider.maxValue = healthComponent.ValueRO.maxHealth;
healthSlider.value = healthComponent.ValueRO.health;
```

Code snippet 4: Separated health logic in ECS

6.3 Threading and Parallelism Support

Traditional object-oriented programming (OOP) approaches in Unity typically operate entirely on the main thread: game logic, physics updates, and rendering are processed sequentially unless explicit multithreading is implemented using native C# threads or Unity's Job System. Integrating threading into an OOP architecture can be complex and error-prone due to shared mutable state, potential race conditions, and the necessity for explicit synchronization mechanisms such as locks or mutexes[3].

In contrast, the Entity Component System (ECS) model within Unity's DOTS (Data-Oriented Technology Stack) is specifically designed for multithreaded execution. ECS organizes

data in contiguous memory structures, and systems execute logic using Unity's C# Job System along with optional Burst compilation for native-level optimizations. This architectural design enables automatic parallelization of entity processing across CPU cores, without requiring developers to write thread-safe code manually [8].

In the implemented prototype, the ECS version automatically distributed combat and movement updates across worker threads, whereas the OOP version remained strictly single-threaded. This contributed to measurable performance improvements in scenarios with high entity counts. The ability to scale with available CPU resources is a key advantage of ECS-based design for large-scale simulations.

6.4 Overhead, Boilerplate and Learning Curve

While ECS offers better performance and architecture for large-scale simulations, it introduces a significant amount of boilerplate code and conceptual overhead, particularly for developers familiar with traditional OOP workflows. Defining components, managing archetypes, and setting up systems require more upfront code and planning compared to the relatively straightforward structure of MonoBehaviour-based OOP.

In the prototype, the ECS and OOP implementations are organized into separate directories. A direct comparison reveals that the ECS version contains approximately **30 script files**, while the OOP version only has **16 script files**. This discrepancy illustrates the additional boilerplate typically associated with ECS architectures, where functionality is divided across multiple components and systems.

For instance, implementing a simple behavior such as unit healing in the OOP version is handled entirely within a single class, `HealerUnit`. In contrast, the ECS implementation distributes the same logic across several files, including `HealerComponent.cs`, `HealthComponent.cs`, and `HealerSystem.cs`. This modular structure promotes separation of concerns and enhances maintainability, but it also increases the amount of setup required for new features.

Moreover, ECS imposes a steep learning curve. Developers must understand data-oriented design, and job safety restrictions. Debugging is also more difficult due to indirection and the absence of typical object references.

7 Conclusion

This thesis explored the design and implementation of a real-time strategy prototype using two distinct architectural approaches in Unity: Object-Oriented Programming (OOP) and Entity-Component-System (ECS). Both versions shared identical game mechanics and logic, allowing for direct comparison in terms of performance, structure, and development patterns.

Performance tests revealed that the ECS implementation significantly outperformed its OOP counterpart in scenarios with large numbers of units. Metrics such as average frame rate, main thread time, and garbage collection frequency consistently favored ECS, demonstrating its advantage in high-scale and compute-heavy simulations. This performance gain was largely attributed to ECS's data-oriented memory layout, multithreading via the Job System, and reduced allocation pressure.

From a maintainability perspective, OOP's reliance on inheritance and stateful objects led to tighter coupling and increased risk of unintended side effects during feature expansion. On the other hand, ECS enabled better decoupling and modularity, which facilitated clean separation of behaviors and more adaptable to feature changes.

However, these benefits came at the cost of increased development complexity. The ECS version introduced more boilerplate, required a deeper understanding of Unity's DOTS framework, and proved more difficult to debug due to its abstracted, data-centric design. In contrast, the OOP model, while less scalable, offered faster prototyping and clearer object relationships, making it more accessible to developers with conventional programming backgrounds.

Ultimately, this study concludes that ECS is well-suited for systems requiring high performance, scalability, and modularity, particularly in games with large numbers of interacting entities like an RTS. However, for smaller projects, rapid prototyping, or teams unfamiliar with data-oriented design, OOP remains a practical and less demanding approach. Choosing between these paradigms should be guided by project scale, performance constraints, and team expertise.

8 Appendix

References

[1] Ian Watson, University of Auckland. AAAI. A Review of Real-Time Strategy Game AI. https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/2478, 2014.

[2] Unity Technologies. *GameObjects and Components*. 2024. https://docs.unity3d.com/Manual/Components.html.

[3] Patryk Galach. Multithreading & Job System in Unity. https://www.patrykgalach.com/2019/07/22/multithreading-job-system-in-unity/, 2019.

[4] Unity Technologies. *Introduction to DOTS*. 2024. https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/index.html.

[5] Timothy Ford, Blizzard. GDC. Overwatch Gameplay Architecture and Netcode. https://www.youtube.com/watch?v=W3aieHjyNvw, 2019.

[6] Mike Acton, Insomniac Games. CppCon. Data-Oriented Design and C++. https://www.youtube.com/watch?v=rX0ItVEVjHc, 2014.

[7] Bob Nystrom. Is There More to Game Architecture than ECS? https://www.youtube.com/watch?v=JxI3Eu5DPwE, 2018.

[8] Toxigon. Boosting Game Performance with Unity's DOTS System. https://toxigon.com/boosting-game-performance-with-unitys-dot-system, 2023.