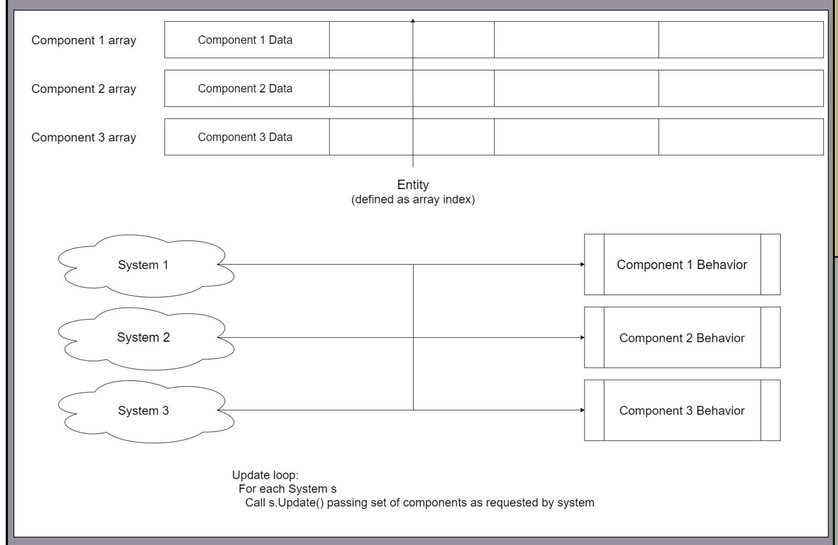
Entity Component System



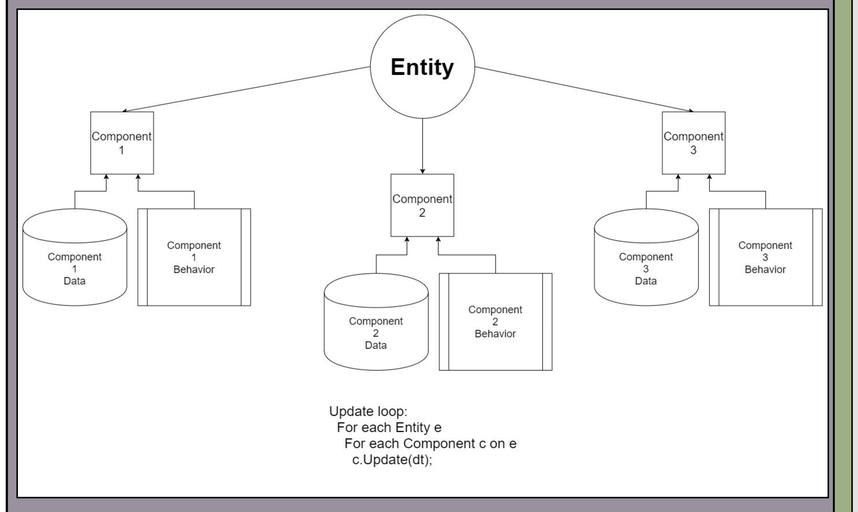
The Entity Component System model has 3 main parts - each word of the name being one of them. The architecture is desirable as it is data oriented and exhibits great performance characteristics as a result. The major downside it has is the way it works isn't very intuitive to non-technical people, who create the majority of content for games.

An **entity** is any "thing" in the game. This could be a player, a mountain, an enemy, or even less tangible constructs like the time remaining in a match. In a "pure" ECS, the entity is just an ID or handle and has different components associated with it.

Components are Plain Old Data (PODs) that collectively contain all the data for each entity. A player entity might have components such as an input source, model, transform, collider, rigidbody, and score.

Systems contain the logic that performs transformations on the data. A system will have specific component requirements for operation, and will execute on any entity that has those components.
A collision detection system might require transform and collider components. It would then detect collisions between all the objects that have both of those components.

Component Centric



This is a more traditional object-oriented style architecture consisting of entities and components similar to those of the ECS model. It seems to be intuitive to non-technical users which makes it particularly desirable in the games industry, but its generally poor cache utilization results in architecture-level performance issues. For these reasons it is popular for smaller scale indie titles made by mostly non-technical teams, but isn't often used in large-scale AAA games which can't afford the performance it.

Entities act as a container in which components are stored. It can receive information and filter it to the correct components. The entity also acts as an intermediary for components that need to access each other.

Components contain both data and logic for updating themselves. They gain access to data they need on other components by going through the parent entity as an intermediary. Components which depend on other components in order to function can have their dependencies checked when they are attached to an entity, ensuring it already has the dependencies before allowing the attachment to succeed.

A Hybrid Approach to Game Engine Architectures

Allan Deutsch and Aaron Kitchen

Hybrid Architecture

In this model, the underlying memory layout is identical to that of the ECS model. To work around the conceptual difficulty non-technical users may have with the architecture, it also supports the coding style used in the component-centric model.

By using SFINAE reflection to detect the presence of update(dt) and other key names from the CC model, a system can be created automatically using a generated lambda. The lambda simply has to iterate over all the stored components of that type invoking their update methods and other optional niceties such as setting a parent pointer.

It also means dependencies are handled in the more elegant manner of an ECS. This enables technical users to write low level systems using data oriented design a la ECS, while still giving non-technical designers the expressiveness and ease of use they desire from the CC model. As an added benefit, it maps the CC model into an ECS memory layout which may result in improved performance over a naïve CC model implementation.

SFINAE example to enable it

```
template<typename T>
struct has_update {
private:
  template<typename>
  struct check : std::true_type
  { };

  template<typename C>
  static auto test( int )->

  check<decltype(
   ::std::declval<C>( ).Update(
     ::std::declval<float>( ))
 ) >;

  template<class>
  static auto test( long )
  ->std::false_type;

  template<typename C>
  struct verify : decltype( test<C>( 0 ) ){};
public:
  static constexpr bool value{ verify<T>( ) };
};

template<typename Component>
ComponentStore<Component>::ComponentStore( ComponentManager& cm ) {
  if constexpr( ::dart::has_update<Component>::value ) {
    cm.RegisterUpdater( [ this ]( float dt ) {
      for( auto &it : m_components ) {
        it.Update( dt );
      }
    } );
  }
}
```

Comparison of Methods

Feature	ECS	CC	Hybrid
Easily mapped to contiguous memory	✓	✗	✓
Simple dependency management	✓	✗	✓
Data-oriented design	✓	✗	✓
Effective way of modeling simulation data	✓	✓	✓
Logic easily extended via embedded scripting language	?	✓	✓
Logical flow makes sense to non-technical people	✗	✓	✓
Allows for encapsulation of data, with ability to update without friend functions	✗	✓	✓

Examples of Client Code

```
// Both of the below examples accomplish the same thing, one using
// each component model

// Both of these examples work for the hybrid model, with the same
// syntax

// Component centric model

class TestComponent final : public Component {
  // For ECS, you'll need to make the updater
  // a friend function, or make this member public
  float m_floatval = 0;
public:
  // For the hybrid approach this doesn't need to be virtual
  void Update( float dt ) override {
    Transform & t = GetOwner().GetComponent<Transform>();
    Console.Log( "Update" );
    t.Position.x += m_floatval;
    m_floatval += dt;
  }
};

// ECS model

void SystemUpdate( float dt, TestComponent& component, Transform& t )
{
  Console.Log( "Update" );
  t.Position.x += component.m_floatval;
  component.m_floatval += dt;
}
```