Introduction to Component Based Design for C++ Games

Sean Middleditch

About Me

- Sean Middleditch
  - DigiPen RTIS Senior
  - Game Projects
    - Subsonic (2010)
    - SONAR (2011)
    - Core (2012)
  - 18 years C/C++ experience
  - http://seanmiddleditch.com

About This Lecture

- Introduction to C++ and OOP
- Component Based Design Overview
- Data Driven Components
- Best Practices

Object Oriented Programming Primer

Object Oriented Programming Primer

- What is OOP?
  - Designing code in functional units
  - Communicating between units with abstract interfaces
  - Encapsulating implementation details
  - Making code comprehensible

Object Oriented Programming Primer

- What is OOP not?
  - Designing code around real-world objects
  - One-to-one mappings of programmer ideas and classes
  - Useless abstractions and excess layers

Object Oriented Programming Primer

- Classes vs Interfaces vs Objects
  - Classes are concrete functionality
    - “A sprite”
  - Interfaces define how different parts of a program communicate
    - “Something that can be drawn”
  - Objects are specific instances of a concept
    - “The hero sprite”

Object Oriented Programming Primer

- Interfaces in C++
  - Defining an interface

```
class IDrawable
{
public:
  virtual IDrawable() {}

  virtual void Draw(RenderContext* ctx) const = 0;

  virtual bool IsVisible() const = 0;
  virtual void SetVisible(bool mode) = 0;
};
```

Object Oriented Programming Primer

- Interfaces in C++
  - Consuming an interface

```
class DrawableSprite : public IDrawable
{
public:
  virtual void Draw(RenderContext* ctx) const;
  virtual bool IsVisible() const;
  virtual void SetVisible(bool mode);

  Sprite* m_Sprite;
  Vec2 m_Position;
};
```

“Classic” Object Oriented Design

Common Misinformation

- Proper OOP design practices have been well understood for several decades
- However, bad practices are frequently used for introductory material
  - … and there's not really much “advanced design” material around
- The result is that there are a lot of programmers who learned how to use OOP totally wrong
- You don't want to be one of them

Real World Objects

- Common Bad Design Practice
  - Almost always the basis for “what is OOP” course material and tutorials
  - Classes that reflect things in real life
    - A car class
  - Interfaces that reflect things people do
    - A gas pedal interface
  - Objects that represent a specific real thing
    - A Ford 2003 Focus object instance

Real World Objects

- Real World Object Complexity
  - Car class, abundance of interfaces
  - Door locking, opening, closing
  - Windows up, down
  - Wipers on, off
  - Brake pressure
  - Gas pressure
  - Steering
  - Stereo
  - Etc.

Real World Objects

- Real World Object Code Proliferation
  - Car class
  - Tank class
  - Motorcycle class
  - Jet class
  - Sailboat class
  - Steam locomotive class

Real World Objects

- Real World Taxonomy
  - Sharing code between “similar” objects
  - Vehicle base class
    - Motorized class
      - Wheel class
      - Jet class
      - Boat class
        - … sailboats are a problem

Real World Objects

- Real World Taxonomy
  - Try number two
  - Vehicle base class
    - LandVehicle class
      - MotorLandVehicle class
      - ManualLandVehicle class
    - NauticalVehicle class
      - MotorboatVehicle class
      - WindPoweredVehicle class
  - “Motorized” code is duplicated

Real World Objects

- Real World Taxonomy
  - The “solution” (not really)
  - Vehicle base class
    - Land functionality
    - Water functionality
    - Motor functionality
    - Sailing functionality
    - Subclasses to select options
      - Car class (enables land, motor)
      - Plane class (enables air, motor)
      - Sailboat class (enables water, sailing)
      - All subclasses have all data/features

Real World Objects

- Real World Taxonomy

Real World Objects

- Real World Taxonomy
  - Creates very large objects with excess data
  - Creates very large interfaces that are hard to document and use
  - Creates huge class diagrams that are hard to explain to new developers
  - Creates a big mess that will slow you down

Good Object Oriented Design

Computer Program Objects

- Computers are all about data and logic
  - Data structures contain information
    - What objects exist in the world
    - What color are the walls
    - How much life does the player have
  - Logic uses data to simulate the game world
    - Objects and textures used to draw world
    - Player and enemy weapon data used in combat

Computer Program Objects

- Computers are all about data and logic
  - Classes and interfaces for data
    - Vehicle body mesh
    - Engine performance characteristics
    - Storage compartment contents and dimensions
  - Classes and interfaces for logic
    - Navigation logic
    - Traction control logic
    - Physical simulation logic

Computer Program Objects

- Focus on abstractions
  - Motor, not CarEngine
  - Navigation, not NauticalCharting
  - Storage, not TruckBed
- Reusable components
  - Motor can be used with a car, train, plane, boat, crane, conveyer belt, bottling machine, elevator, automatic door, ventilation system, etc.

Computer Program Objects

- Classes in C++
  - Think about what they DO, not what they ARE
    - Button, Lever, Switch, and PressurePlate all trigger actions, so just make one Trigger class

Computer Program Objects

- Classes in C++
  - Think about what they DO, not what they ARE
    - Button, Lever, Switch, and PressurePlate all trigger actions, so just make one Trigger class
  - Think about what they CONTAIN
    - PlayerDude, BulletIcon, and EnemyFace all have an image, so just make one Sprite class

Computer Program Objects

- Classes in C++
  - Think about what they DO, not what they ARE
    - Button, Lever, Switch, and PressurePlate all trigger actions, so just make one Trigger class
  - Think about what they CONTAIN
    - PlayerDude, BulletIcon, and EnemyFace all have an image, so just make one Sprite class
  - Think about how they are USED
    - World and Inventory both contain objects, but are used very differently, so make them separate (and thus smaller and simpler)

Aggregation vs Inheritance

Inheritance in C++

- Inheritance allows C++ to “extend from” the data and logic of a class
  - Use this when you have an extended interface
  - Logger can be used anywhere, but FileLogger adds the ability to write to files
  - Almost always has virtual methods
  - Read as “is-a”
    - FileLogger is a Logger

Inheritance in C++

- Inheritance allows C++ to “extend from” the data and logic of a class

```
class Logger
{
public:
  virtual void PrintMessage(std::string message);
};

class FileLogger : public Logger
{
public:
  void OpenFile(std::string filename);
  virtual void PrintMessage(std::string message);
};
```

Inheritance in C++

- Inheritance can “add to” a class
  - Backwards design
  - Doesn't read as “is-a”
    - “Enemy is a Health” doesn't seem right
  - Can cause conflicts
  - Hard to document
  - Just don't do it

Inheritance in C++

```
class Health
{
public:
  int m_CurrentHP;
  int m_MaxHP;
};

class Attack
{
public:
  int m_WeaponSpeed;
  int m_WeaponDamage;
};

class Enemy : public Health, public Attack
{
public:
  std::string m_Name;
};
```

DON'T DO THIS!

Aggregation in C++

- Aggregation is containment
  - Reads as “has” or “has-a”
    - “Enemy has Health”
    - “Enemy has an Attack”
    - Much better
  - Treat objects as data instead of structure
  - Simpler to document
  - Way less problems

Aggregation in C++

```
class Health
{
public:
  int m_CurrentHP;
  int m_MaxHP;
};

class Attack
{
public:
  int m_WeaponSpeed;
  int m_WeaponDamage;
};

class Enemy
{
public:
  Health m_Health;
  Attack m_Attack;
  std::string m_Name;
};
```

Component Based Design

Component Based Design

- Use aggregation for everything
  - A GameObject is not a Sprite
    - Game objects *have* a Sprite
  - A GameObject is not a Physics
    - Game objects *have* Physics
  - The Engine is not a Graphics
    - Engines *have* Graphics

Component Based Design

- More than just aggregation
  - Component based design is a way of thinking about designing systems
  - Aggregation is a way of building a class
  - Component based design is a way of building an entire software application

Component Based Design

- Components make up larger systems
  - Components are self-contained pieces of functionality or data that are used to make up a whole
  - A game object has many pieces
    - Visual representation
    - Physics data
    - AI logic
    - Location in world
  - Each of these pieces is a separate component

Component Based Design

- Components are separate from each other
  - Graphics does not care about physics
  - Physics does not care about graphics
  - They share the Transform (position, rotation, scale)
  - Three components
    - One for graphics data
    - One for physics data
    - One for shared transform data

Component Based Design

- Components are decoupled
  - Graphics does not care about AI
  - You can add or remove AI to a game object without affecting graphics at all
  - Different game objects can have different sets of components
    - The player doesn't have an AI component
    - Enemies don't have a Controller component
    - Items don't have AI or Controller components
    - Triggers don't even have a graphics component

Component Based Design

- Communicate along well-defined interfaces
  - Physics component has an interface
    - SetVelocity(x, y)
    - Update(dt)
    - SetMass(kg)
    - GetMass()
  - Controller component uses that interface
    - Player moves forward, Controller component sets a forward velocity on Physics component

Component Based Design

- Prefer smaller, simpler interfaces
  - Larger interfaces can be split up
  - “Physics” is two different things
    - Collision detection
    - Collision resolution and integration

Component Based Design

- Prefer smaller, simpler interfaces
  - These are two different components
    - Collider component
    - Physics component
  - Don't always need both
    - Triggers need to know if they collide with the player, but can't be pushed around and are not affected by gravity

Component Based Design

- Prefer smaller, simpler interfaces
  - Simpler Collider interface
    - IsColliding(GameObject* other_object)
  - Can swap out different implementations
    - BoxCollider vs CircleCollider
    - Both have the same interface
      - Using “good” inheritance in C++
  - Triggers don't care about how they're colliding
  - They only care that they are colliding

Component Based Design

```
class Collider
{
public:
  virtual bool IsColliding(GameObject* object) const = 0;
};

class BoxCollider : public Collider
{
public:
  virtual bool IsColliding(GameObject* object) const;
  AABB m_Box;
};

class CircleCollider : public Collider
{
public:
  virtual bool IsColliding(GameObject* object) const;
  float m_Radius;
};
```

Data Driven Design

Data Driven Design

- Using data to control logic
  - i.e., read a list of commands from a text file
  - This can include commands for building objects
    - Lists of components and data

```
Enemy {
  Transform { scale = 2 }
  Health { max = 10 }
  Attack { speed = 2, damage = 5 }
  BoxCollider { width = 2, height = 3 }
  Physics { mass = 10, use_gravity = true }
  AI { script = "chase_player" }
  Sprite { image = "mean_dude.png" }
}
```

Data Driven Design

- Easily build new objects from existing components

```
Trap {
  Transform { scale = 1 }
  BoxCollider { width = 4, height = 4 }
  Trigger { script = "hurt_player", fire_once = true }
}

Coin {
  Transform { scale = 1 }
  CircleCollider { radius = 1 }
  Physics { mass = 1, use_gravity = true }
  Trigger { script = "give_coin", fire_once = true }
  Sprite { image = "shiny_penny.png" }
}
```

Data Driven Design

- Reduces iteration time
  - Designers don't need to wait on programmers for every change
  - Does not require a recompile of the game
- Use it everywhere you can
  - Remove as much data as possible from the code

Coding Components in C++

Components in C++

- Actually using components requires some work
  - This requires some in-depth knowledge of C++
  - Not hard stuff, but it may all be new to you
  - Topics include:
    - Inheritance
    - Virtual functions
    - Standard containers
    - Object lifetime management
    - Constructors and destructors

Component in C++

```
class GameObjectComponent
{
public:
  virtual std::string GetName() const = 0;
  virtual void Update(double dt) = 0;

  GameObject* m_Parent;
};

class GameObject
{
public:
  ~GameObject();
  void AddComponent(GameObjectComponent* cmp);

  std::map<std::string, GameObjectComponent*> m_Components;
};
```

Components in C++

- GameObjectComponent is an interface
  - Every component has a name (Sprite, Transform, etc.)
  - Every component can be updated each frame
    - (actually not that great of a design, but it fits on a slide)
  - “Pure virtual” methods define interface
- GameObject is just a container of components
  - It aggregates other objects to make a more interesting whole

Components in C++

```
class GameObjectComponent
{
public:
  virtual std::string GetName() const = 0;
  virtual void Update(double dt) = 0;

  GameObject* m_Parent;
};

class GameObject
{
public:
  ~GameObject();
  void AddComponent(GameObjectComponent* cmp);
  GameObjectComponent* GetComponent(std::string name);

  std::map<std::string, GameObjectComponent*> m_Components;
};
```

Components in C++

- Sprite and Trigger are specific components
  - They implement the same component interface
    - Can ask for their name and update them
    - They do not require any special code to interact with them
    - Yet they do totally different things
  - In C++ terms, they derive from a common base class and override virtual methods

Components in C++

- Add a component to an object
  - Really simply
  - Basic use of STL containers
  - Inserts component into map, sets parent

```
void GameObject::AddComponent(GameObjectComponent* component)
{
  m_Components[component->GetName()] = component;
  component->m_Parent = this;
}
```

Components in C++

- Finding components
  - Basic (but really ugly) use of STL containers
  - Checks if component exists and returns it

```
GameObjectComponent* GameObject::GetComponent(std::string name)
{
  std::map<std::string, GameObjectComponent*>::iterator i =
      m_Components.find(name);
  if (i == m_Components.end())
    return NULL;
  else
    return i->second;
}
```

Components in C++

- Updating components
  - Basic (but really ugly) use of STL containers
  - Forwards a method call to all components

```
void GameObject::Update(double dt)
{
  for (std::map<std::string, GameObjectComponent*>::iterator
       i = m_Components.begin(), e = m_Components.end();
       i != e; ++i)
  {
    i->second->Update(dt);
  }
}
```

Components in C++

- Clean up after yourself
  - GameObject owns its components
  - It's responsible for freeing their resources

```
GameObject::~GameObject()
{
  for (std::map<std::string, GameObjectComponent*>::iterator
       i = m_Components.begin(), e = m_Components.end();
       i != e; ++i)
  {
    delete i->second;
  }
}
```

Data-Driven Components

Data-Driven Components

- Real power of components comes from data
  - Ability to dynamically create objects from text files
  - Needs a text file reader/parser
    - TinyXML
      - http://www.grinninglizard.com/tinyxml2/index.html
    - Jansson
      - http://www.digip.org/jansson/
    - IO Streams
      - Standard C++ library

Data-Driven Components

- Mapping text to code
  - Text file will specify names of components
  - Text file will specify component data
  - Need a method to create a component from text name
  - Need a method to popular component fields from text values

Data-Driven Components

- Factory pattern
  - A factory is a method for creating objects based on an identifier
  - Often implemented as a simple switch statement or a series of if/else statements
  - More complicated and flexible factories are possible, but totally unnecessary in most cases

Data-Driven Components

- Component factory
  - Add an “if” statement for each component

```
GameObjectComponent* ComponentFactory(std::string name)
{
  if (name == "Sprite")
    return new SpriteComponent();
  else if (name == "Trigger")
    return new TriggerComponent();
  else
    return NULL;
}
```

Data-Driven Components

- Reading component data
  - Components have member variables
    - e.g., SpriteComponent has an image property
  - Need to convert from text to these values
  - Called “deserialization” or “unmarshalling”
    - Convert to a file is “serialization” or “marshalling”
  - Requires a new interface method in GameObjectComponent

Data-Driven Components

- Add a new set of methods to the interface

```
class GameObjectComponent
{
  //...
  virtual void Deserialize(Reader* reader) = 0;
  //...
};

class SpriteComponent : public GameObjectComponent
{
  //...
  virtual void Deserialize(Reader* reader);
  //...
};
```

Data-Driven Components

- Reader is an interface
  - It has a ReadNext() method
  - What or how it reads is irrelevant
    - Maybe it reads XML, maybe JSON, maybe binary, maybe some custom text protocol
    - Components don't need to know or care
  - Actual implement might be XmlReader
    - Same interface no matter which implementation
    - We only care about what it DOES, not what it IS

Data-Driven Components

- Implement Deserialize for each component
  - Add a new “if” statement for each property
  - Simple, unsophisticated, works just fine

```
void SpriteComponent::Deserialize(Reader* reader)
{
  std::string name, value;
  while (reader->ReadNext(name, value))
  {
    if (name == "image")
      m_Image = value;
  }
}
```

Component Design

Component Design

- Logic or Data?
  - Components can contain logic, like the Update() method
  - Components contain data used by logic
  - Some components are just data, like Transform
  - Some components are just logic, like AI
  - Some components can be both, like Sprite
    - It contains data about the sprite to draw
    - It can have the logic to draw... but should it?

Component Design

- Components should be used everywhere
  - That means that the game engine is made up of components, too
    - No, not GameObjectComponents, but EngineComponents, i.e. Systems
    - PhysicsSystem, GraphicsSystem, InputSystem, LogicSystem, etc.
  - Should these be primarily logic or data?
    - Does GraphicsSystem do any drawing, or does it just contain scene data for game objects?

Component Design

- Use logic in engine systems
  - All physics logic is done by PhysicsSystem
  - All rendering is done by GraphicsSystem
- Use only data in game object components
  - SpriteComponent just references a sprite
  - PhysicsComponent just has mass and velocity
- Systems manipulate game objects
  - Iterate list of game objects, or lists of components

Component Design

- Advantages to keeping logic in systems
  - Control over ordering
    - Ensuring all physics updates happen before all graphics updates
  - Efficiency
    - No need to call Update() on objects that don't have any components that need updating
  - Easier to handle more complex inter-object logic needs
    - Sorting sprites for rendering, etc.

Component Design

- Intrusive linked lists are your friends
  - Each component class has next and previous members
  - Each system has a Register/Unregister pair of methods to register their relevant components
  - Components register themselves with systems
  - Systems can iterate over all relevant components easily

Component Design

```
class SpriteComponent : public GameObjectComponent
{
  //...
  SpriteComponent* m_Previous;
  SpriteComponent* m_Next;
  //...
};

SpriteComponent::SpriteComponent()
{
  g_Engine->m_GraphicsSystem.Register(this);
}

SpriteComponent::~SpriteComponent()
{
  g_Engine->m_GraphicsSystem.Unregister(this);
}
```

Component Design

```
class GraphicsSystem
{
public:
  SpriteComponent* m_Head;
};

GraphicsSystem::Register(SpriteComponent* cmp)
{
  cmp->m_Next = m_Head;
  cmp->m_Previous = NULL;
  m_Head = cmp;
}

GraphicsSystem::Unregister(SpriteComponent* cmp)
{
  // seriously, you know how to do this
}
```

More Component Usage

More Component Usage

- There are many places to use components
- AI Behaviors
  - Build complex intelligent agents out of small, simpler behaviors
    - ChaseBehavior
    - FleeBehavior
    - GuardBehavior
    - PatrolBehavior
    - FindCoverBehavior

More Component Usage

- User Interface Controls
  - ClickableControl
  - ScrollableControl
  - DraggableControl
  - TextControl
  - IconControl
  - ContainerControl

More Component Usage

- Particle effect system
  - SpawnEffect
  - GravityEffect
  - TimeEffect
  - DistanceEffect
  - ForceEffect
  - InterpolateEffect

Closing Thoughts

Closing Thoughts

- Components are a tool
  - You don't need to use every tool
  - For simpler games, the costs of adding components might outweigh the benefits
  - Thousands and thousands of games have been written, shipped, and loved, all without components
  - Same goes for other tools (scripting, metadata, editors, even C++)

Additional Resources

- Evolving Your Hierarchy (Mick West)
  - http://cowboyprogramming.com/2007/01/05/evolve-
- Game Object Structure: Inheritance vs Aggregation (Kyle Wilson)
  - http://www.gamearchitect.net/Articles/GameObjects
- Game 200 Lectures (Chris Peters, Ben Ellinger)
  - DigiPen Institute of Technology, GameCentral on Moodle

Questions?

```
std::queue<Question> m_QuestionQueue;

void EndLecture()
{
  while (!m_QuestionQueue.empty())
  {
    Answer(m_QuestionQueue.front());
    m_QuestionQueue.pop();
  }
}
```