Techniques and Strategies for Data-driven design in Game Development

Scott Shumaker
Outrage Games

Motivations

- Many games are never completed
- Many completed games are not financially successful
- These failures are frequently caused by flaws in the development process

Game development is hard

- From a technology standpoint, game development is no more difficult than other software development
- However, properly functioning technology is just one of many requirements for a successful title
- The majority of the development time is not spent developing technology, but content and gameplay

Content

- Art
- Levels
- Music
- Sound FX
- Scripting
- Cutscenes
- Basically anything that isn't code

Gameplay

- Rules
- Goals
- Controls
- AI and object behavior
- Overall play mechanics

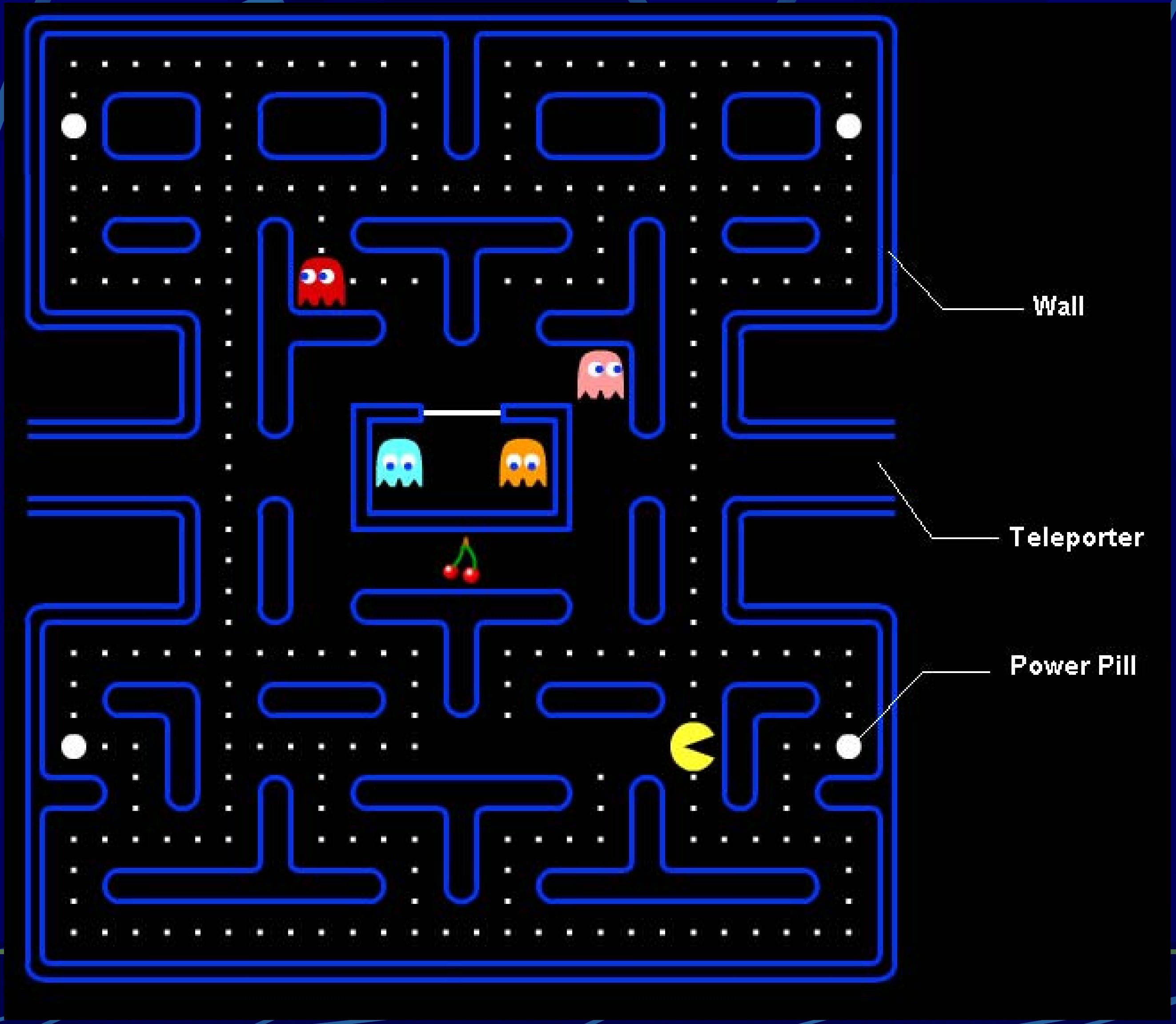
Fun and Appeal

- Making a game fun and appealing is one of the hardest challenges in game development
- Difficult to predict how a decision will impact fun factor until after it is actually implemented
- Gameplay and content both require a large amount of tweaking before the game looks, feels, and plays right

Fun and Appeal - Summary

- Fun and appeal are huge factors in a game's commercial success
- The difficulty in achieving and assessing these important traits make them the biggest risk in the project
- Improving our process for adjusting gameplay and tweaking content will help us minimize risk

Case Study: PacMan



What makes PacMan fun?

- Level design
  - Location of walls
  - Location of dots
  - Location of power-pills
- Enemy behavior
  - Speed (before and after power pills)
  - Respawn rate
  - AI
- Player behavior
  - Speed
  - Invincibility time

Tweaking enemy and player speed, 1st attempt

- Pac-man was designed a long time ago
- They probably hard-coded the speed

```
// player.cpp
void player_move(int direction, player* pplayer)
{
   int pixels_to_move = 17; // player moves 17 pixels / frame

   if (direction == DIR_RIGHT)
   {
      pplayer->pos.x += pixels_to_move;
   }
   // etc
}


// enemy.cpp
void enemy_move(int direction, enemy* penemy)
{
   int pixels_to_move = 17; // enemy moves 17 pixels / frame

   if (direction == DIR_RIGHT)
   {
      penemy->pos.x += pixels_to_move;
   }
   // etc
```

Problems with 1st approach

- Player and enemy have the same speed, so you need to change it in multiple places to tweak it
- We need to recompile and re-run the game to see the results of our change
- Only someone familiar with this specific code can tweak it

Tweaking speed, 2nd approach

- Pull the variables out and place them in a separate file
- You can do this with a #define

```
// game_constants.h
#define PLAYER_AND_ENEMY_SPEED 17


// player.cpp
void player_move(int direction, player* pplayer)
{
   int pixels_to_move = PLAYER_AND_ENEMY_SPEED;

   // etc
```

- Or even better, with a const global variable

```
// game_constants.h
extern const int PLAYER_AND_ENEMY_SPEED;

// game_constants.cpp
const int PLAYER_AND_ENEMY_SPEED = 17;

// player.cpp
void player_move(int direction, player* pplayer)
{
   int pixels_to_move = PLAYER_AND_ENEMY_SPEED;

   // etc
```

Problems with 2nd approach

- We still need to recompile and re-run to see our changes
- You still can't expect a designer to be able to make changes
- Different, interrelated variables may still be spread across the code

Tweaking variables

- Put all of the variables in a text file, excel table or database to keep track of them
- Designers or programmers can both edit this ‘data source’
- Now, we could manually update the variables in the code whenever they change in the data source
- Instead, why not enable our program to read in the data source itself?

Advantages of moving data externally

- We no longer need to recompile to tweak variables
- Designers and programmers can edit the data source
- You can view all of the variables at once
- You can even change these variables on a per-level basis
- You can write more intuitive tools to edit the variables

Extending the data source

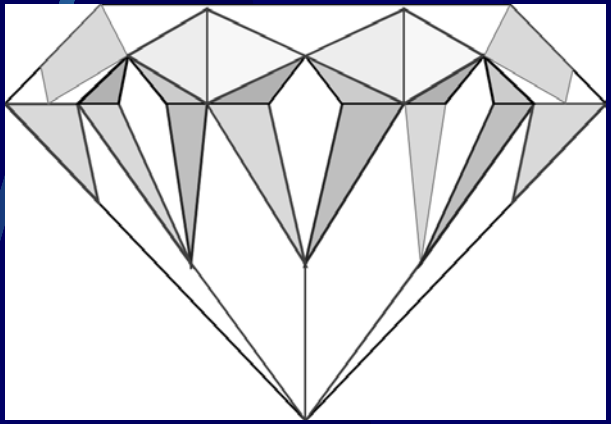
- In PacMan, you can apply this technique to more than just numerical variables
- Levels can be stored as some kind of data file
- You can write a level editor that will allow graphical editing and then write out to this data format
- The game can then read in the data file
- You can extend this technique to virtually any aspect of gameplay or content
  - Character art (models / sprites)
  - Visual effects
  - Spawning of enemies / powerups
  - Attacks, damage, health
  - Etc.

Case study Summary

- Code is a poor location for behavior that needs to be changed frequently
- Moving game behavior out of code and into data results in a more efficient process for tweaking content and gameplay
- We call this technique Data-Driven Design
- This lecture will discuss techniques to achieve good data-driven design

Part II - Data-driven design methodologies

Technique I: Separate Hard and Soft Architecture



Hard Architecture

- Hard Architecture is code that is low-level and would be pretty uniform across different titles
- Examples:
  - Renderer (graphics system)
  - Sound system
  - Low-level input
  - Resource loader (I/O)
  - Low-level utility classes
  - Math library

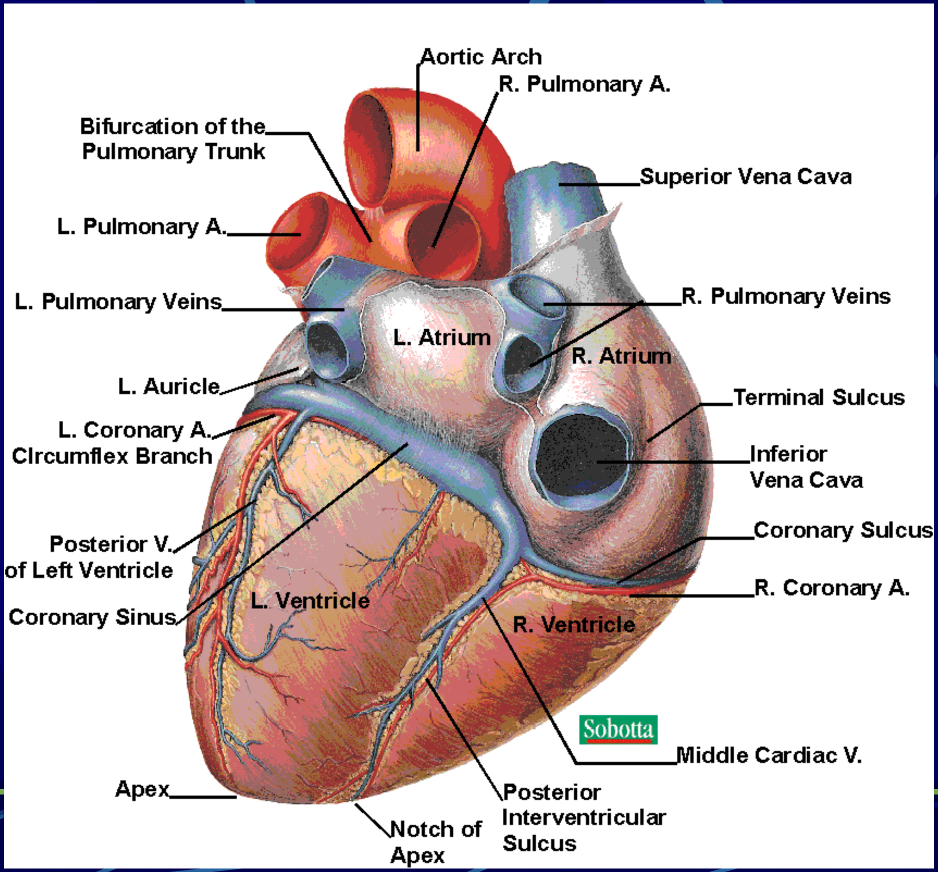
Soft architecture

- Soft Architecture is code that is specific to the game that you are working on
- Examples:
  - Powerup spawning
  - Inventory system
  - Combat system
  - Scoring system

Hard and Soft Architecture

- The distinction is very important
- Soft architecture is allowed to know about the hard architecture systems, but not vice versa
- Only be your soft architectural systems that are responsible for retrieving and manipulating game data
- At some point, they might pass data to the hard architecture low-level systems

Technique II: Separate functionality into Systems



Tokens

- Every discrete element of the game is a token
- Example: Pac-man
  - Dots
  - Power pills
  - Ghosts
  - Pac-man
  - The ghost respawner
  - The level itself

All of these are tokens, or game elements

The systems approach

What is a game system?

- A self-contained module
- Operates on some aspect of your game tokens
- Usually performs a piece of game functionality and implement a specific type of game behavior
- Should be able to be tested in isolation from most other systems

Case Study: Smash TV 3D

- The final 494 project that I and two other programmers worked on
- Designed as a modern, 3d version of the classic SNES and arcade game Smash TV
- Very fast-paced action game filled with hordes of enemies, weapons, and powerups

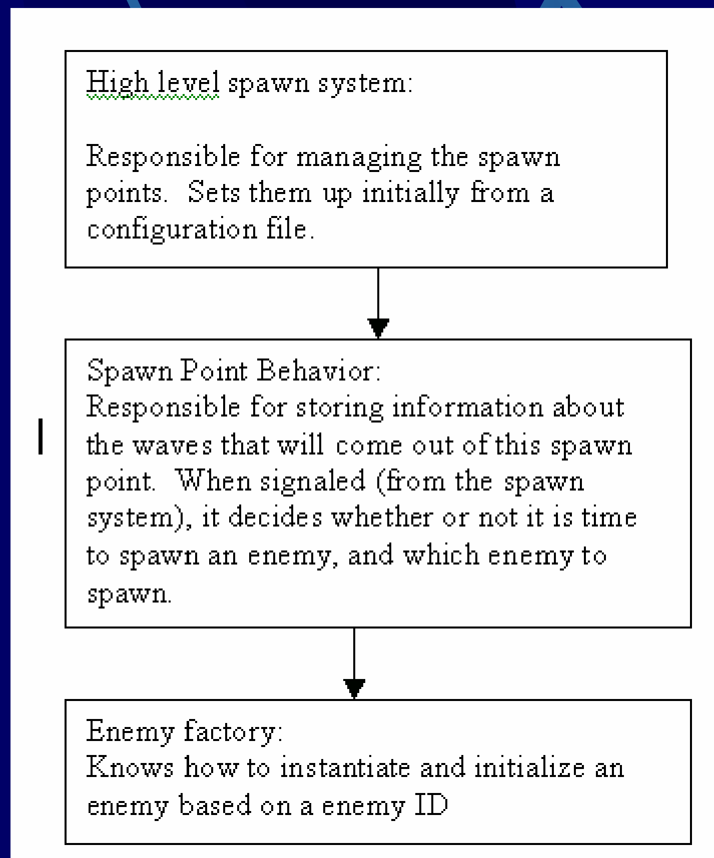
Case Study: Smash TV 3D

- Goals of enemy spawning system
  - Frenetic action
  - Responsive to player
  - Not too difficult, not too easy
  - Adjustable based on player difficulty
  - Good complement of enemies

Smash TV 3D Spawn System

- Spawn points – placed at various points in the level (usually behind the doors)
- Each spawn point is associated with a number of waves
- Each wave is a group of enemies (number, configuration, and delay between each enemy)

Spawn System 2



Spawn System Analysis

- Behavior of the spawning system is very game specific
- The spawning system operates on spawn points, waves, and enemies
- It has sole responsibility for managing waves and spawn points in the game, but for enemies, it only manages their creation

The Systems Approach

- Game systems manage the tokens in your game world
- Design your systems around functional aspects of your game, not around the tokens themselves
- You want each system to be responsible for a specific game behavior, and you'll find that your tokens end up interacting with multiple game systems

Bad example #1

Grunt system

Idea: A system that manages all aspects of the “grunt” enemies, including creation, initialization, behavior / AI, collisions, projectiles, death, rendering.

Why it sucks:

A large portion of the functionality in your grunt system will need to be duplicated for all your other enemies, which will also require systems under this design

Bad Example #2

Enemy System

Idea: A system that manages all aspects of all enemies

Why it bites:

You soon discover that the player also shares a lot of common functionality with enemies, like rendering, weapons, collision detection, etc.

Systems Approach Strategies

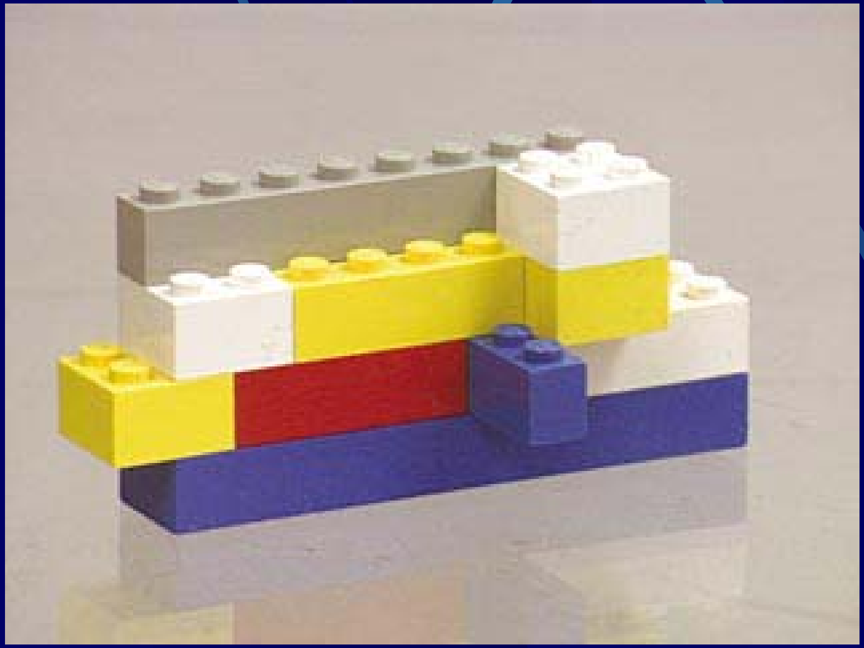
- Break up your systems by functionality
- In the previous examples, we might have separate systems for:
  - Creation
  - Weapons
  - Behavior / AI
- A good rule of thumb: If there are some tokens in the world that could benefit from some of a system’s functionality, but they cannot use it because of other functionality in the system, it should be split

Systems Approach Summary

- Functionality common to different objects is all in one place.
- Each system needs only limited knowledge about various tokens. This reduces coupling.
- If the system works with one kind of object, it will work with all kinds of objects that have the same aspect (this makes it easier to test)

You can view each token as a set of parameters to one or more systems. This allows you to tweak the tokens themselves to change the way your game works.

Technique III: Component-based Architecture



Token Architecture

- Most games have many different types of tokens
- Many tokens share some properties but differ in others
- Example: Two different enemies
  - Both have ‘health’
  - Both have ‘score’
  - Probably have different AI
  - May have different movement types (e.g. flying or walking)

Token Architecture

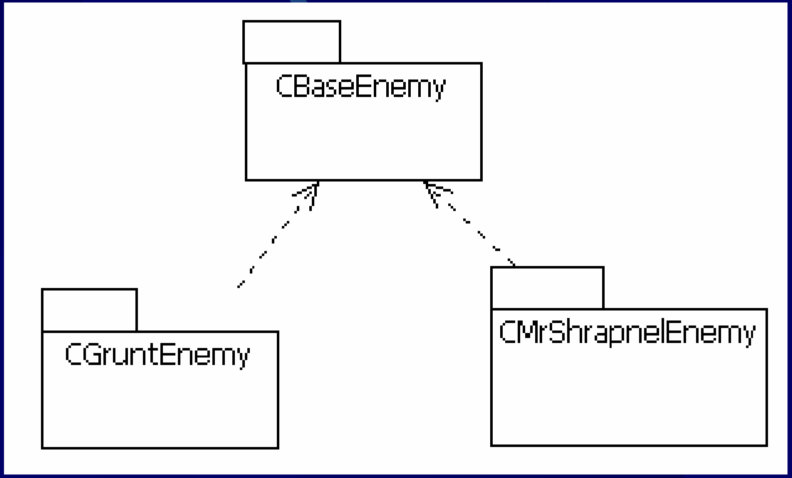
- We want to practice code and interface re-use for tokens
  - Copy and paste code hurts maintainability
  - Identical functionality in different tokens should be treated uniformly by the system that manages that functionality
- How do we structure our tokens to best realize code and interface re-use?

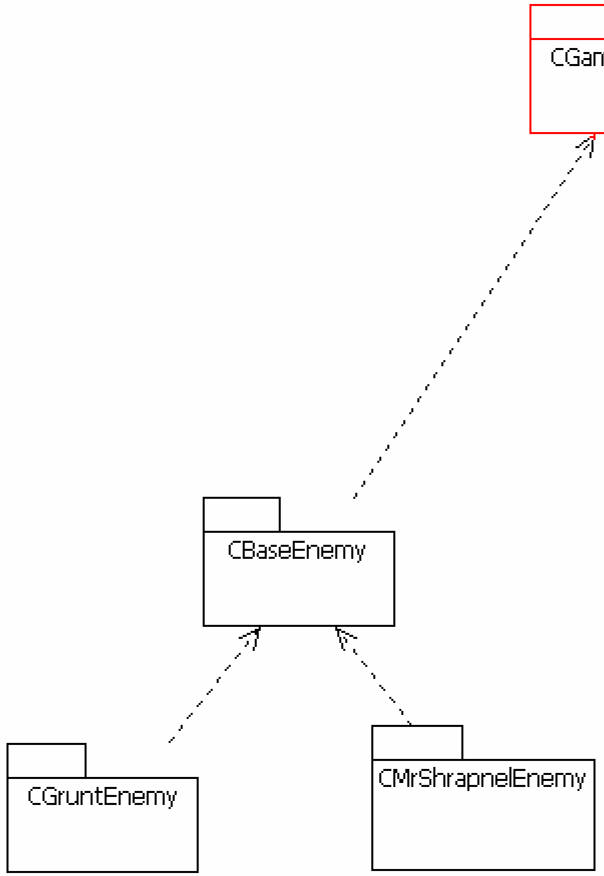
Approach I: Inheritance-based class hierarchy

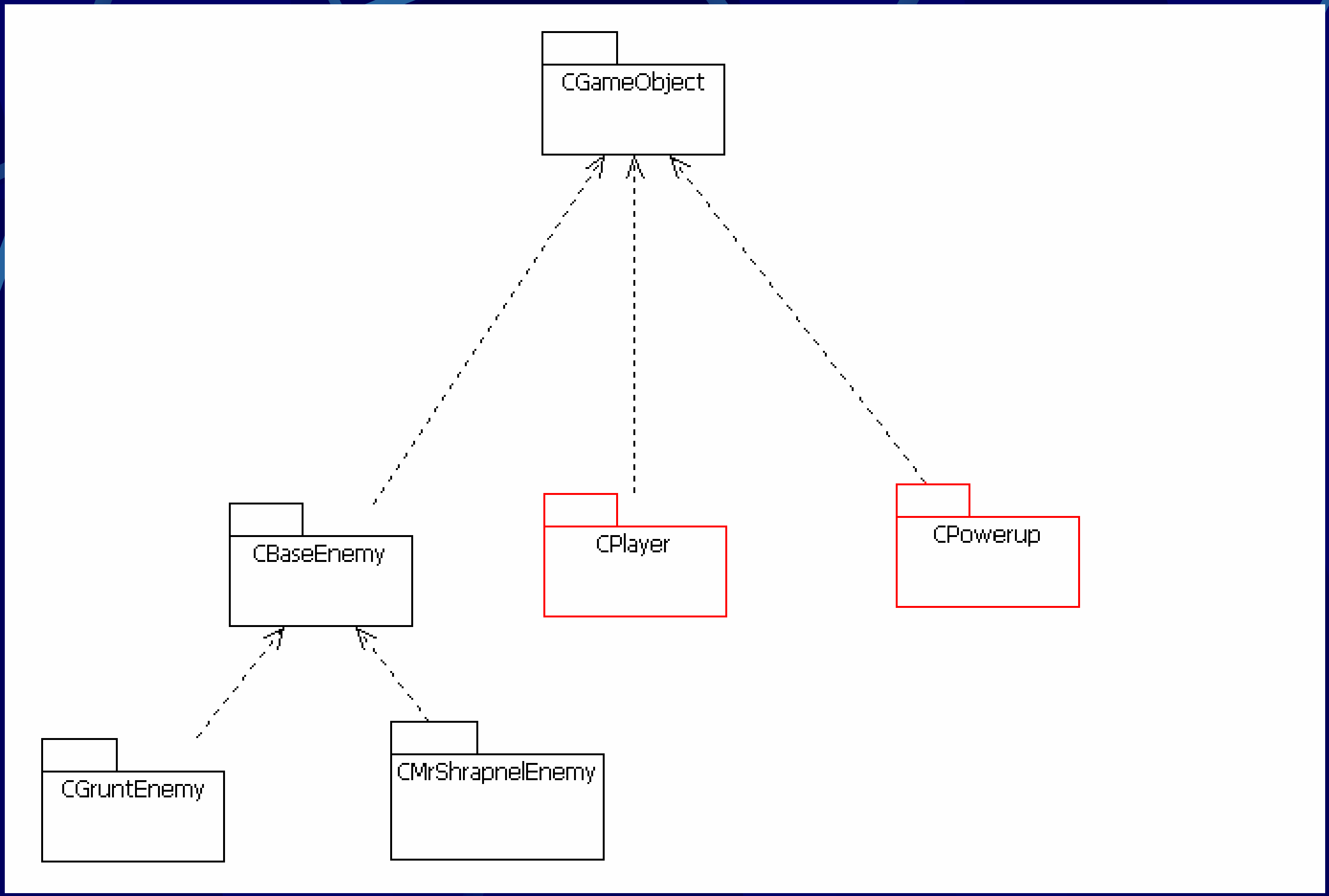
- We can use C++ inheritance to share functionality and interface

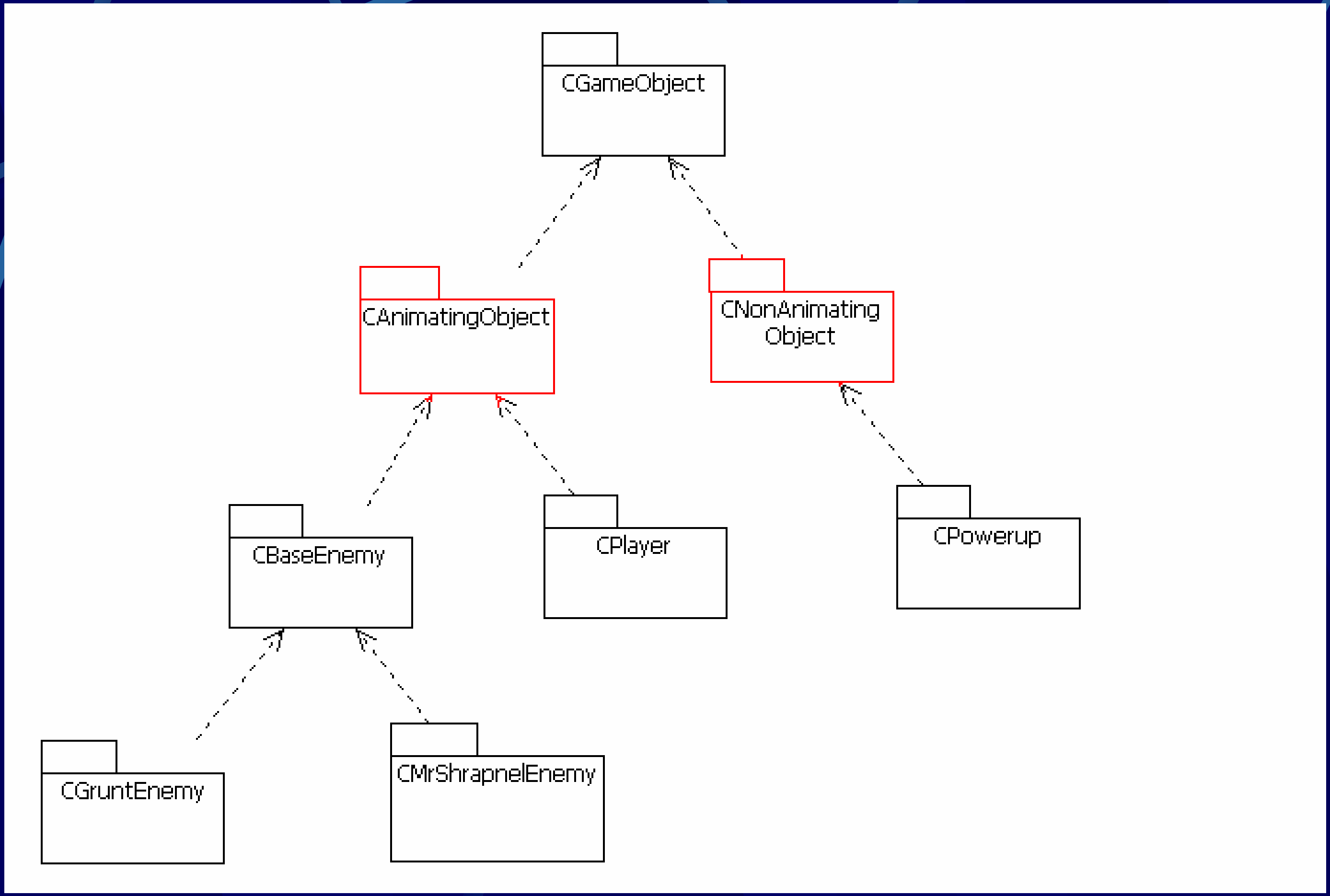
```
class CBaseEnemy
{
    int m_health;
    int m_score;

    // etc.
};

class CGruntEnemy : public CBaseEnemy
{
    // etc.
};

class CMrShrapnelEnemy: public CBaseEnemy

// etc.
```

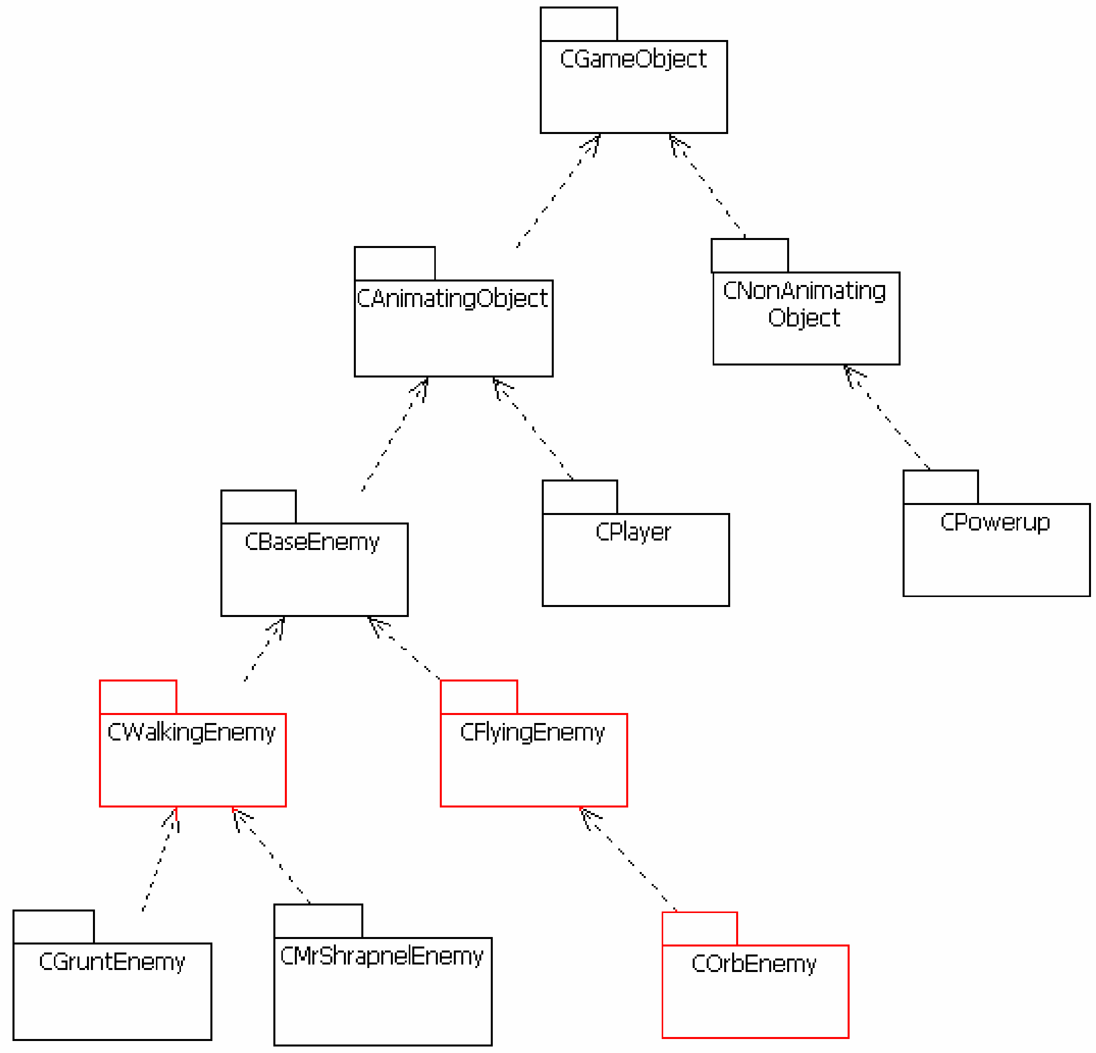
- The inheritance diagram looks like this:

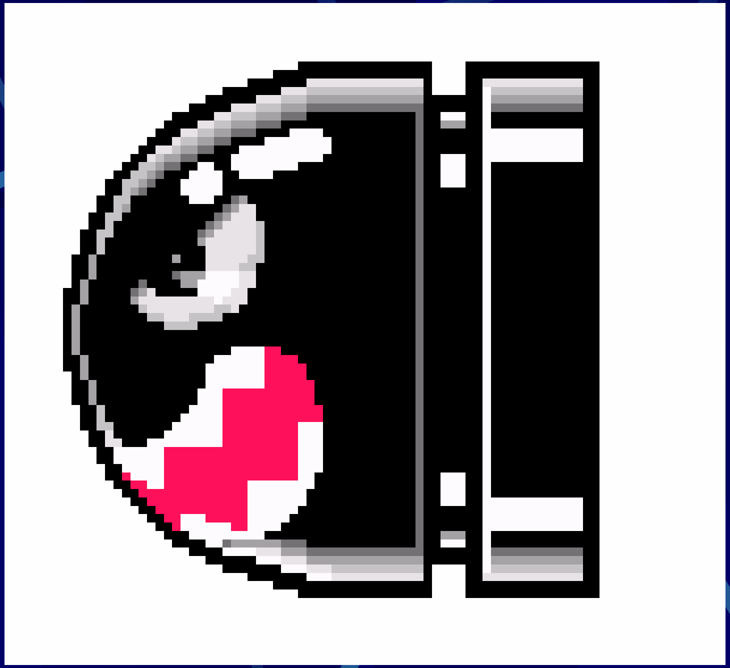



CGameObject
CBaseEnemy
CGruntEnemy
CMrShrapnelEnemy

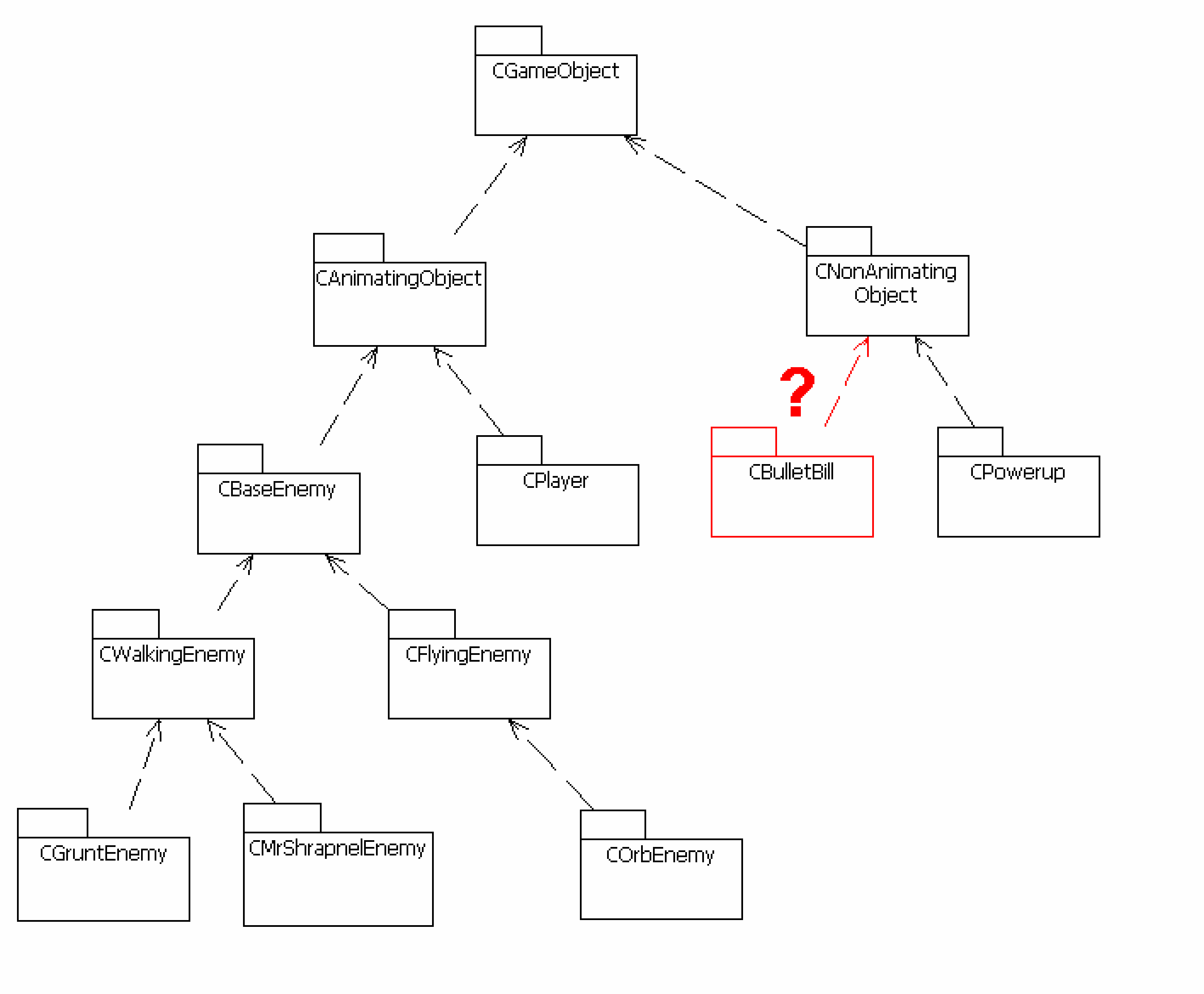

CGameObject
CBaseEnemy
CPlayer
CPowerup
CGruntEnemy
CMrShrapnelEnemy

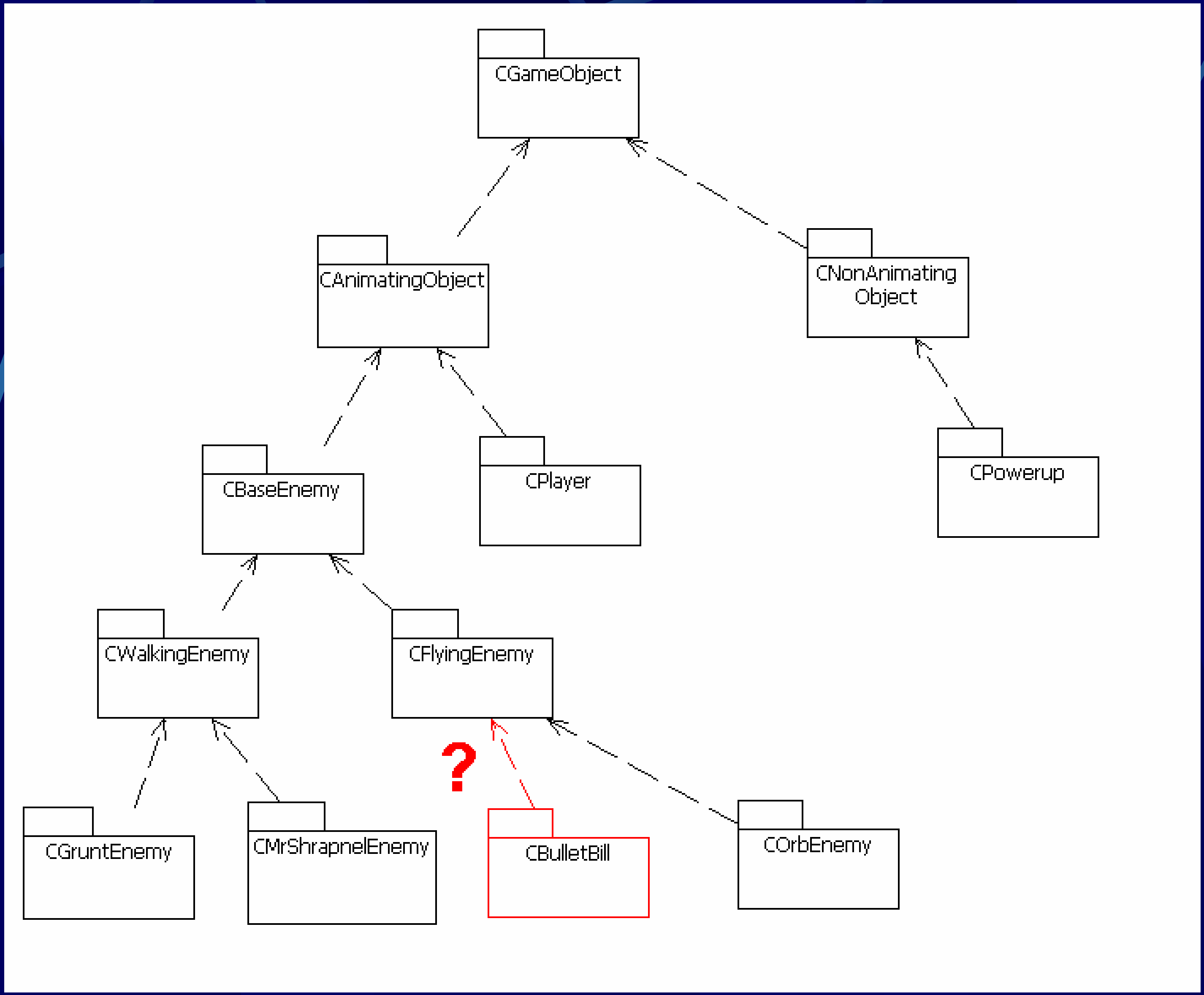

CGameObject
CAnimatingObject
CNonAnimating
Object
CBaseEnemy
CPlayer
CPowerup
CGruntEnemy
CMrShrapnelEnemy


CGameObject
CAnimatingObject
CNonAnimating Object
CBaseEnemy
CPlayer
CPowerup
CWalkingEnemy
CFlyingEnemy
CGruntEnemy
CMrShrapnelEnemy
COrbEnemy

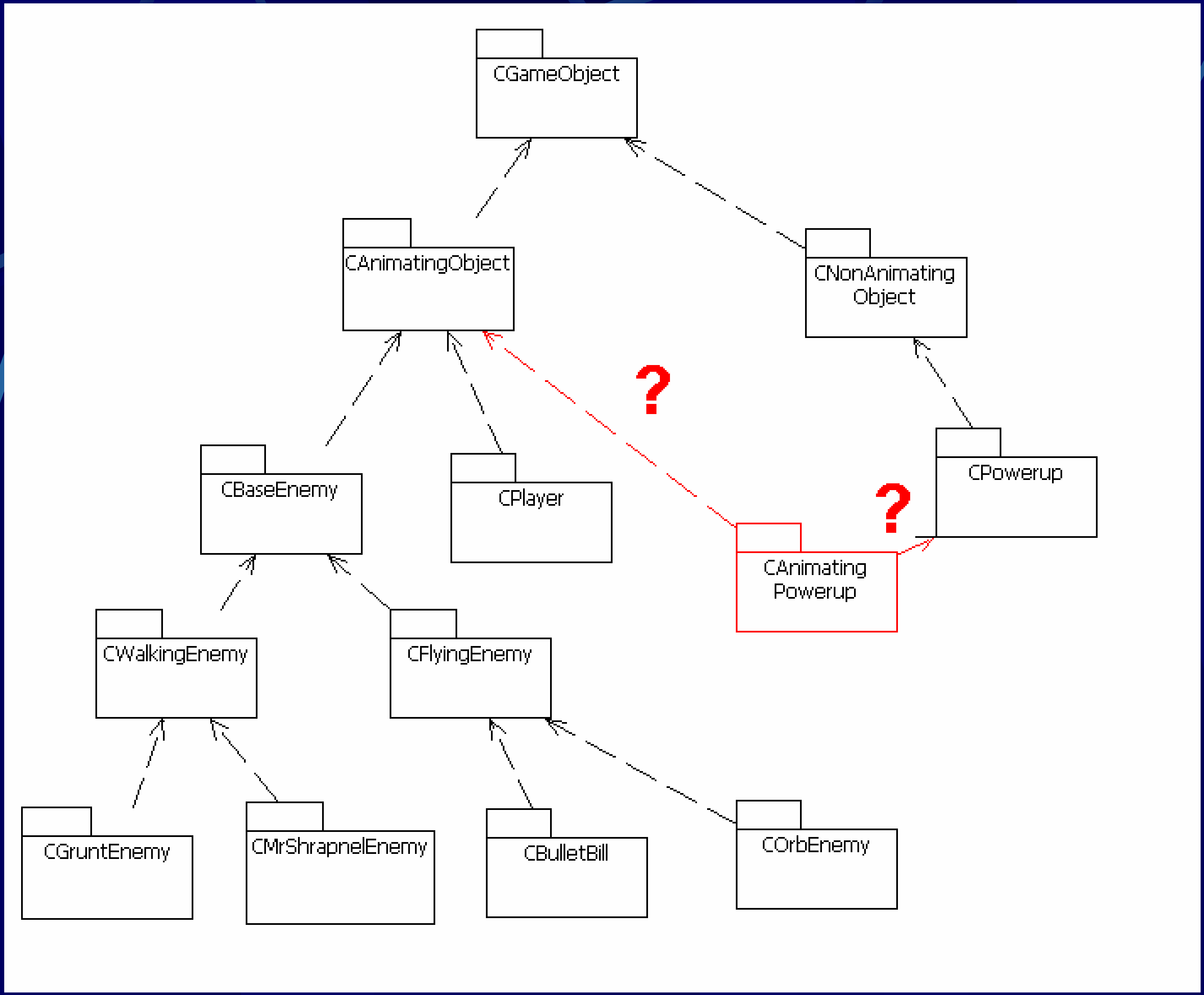


- Bullet Bill:
  - Enemy
  - Flying
  - Non-animating


CGameObject
CAnimatingObject
CNonAnimating Object
CBaseEnemy
CPlayer
?
CBulletBill
CPowerup
CWalkingEnemy
CFlyingEnemy
CGruntEnemy
CMrShrapnelEnemy
COrbEnemy



CGameObject
CAnimatingObject
CNonAnimating Object
CBaseEnemy
CPlayer
CPowerup
CWalkingEnemy
CFlyingEnemy
?
CGruntEnemy
CMrShrapnelEnemy
CBulletBill
COrbEnemy



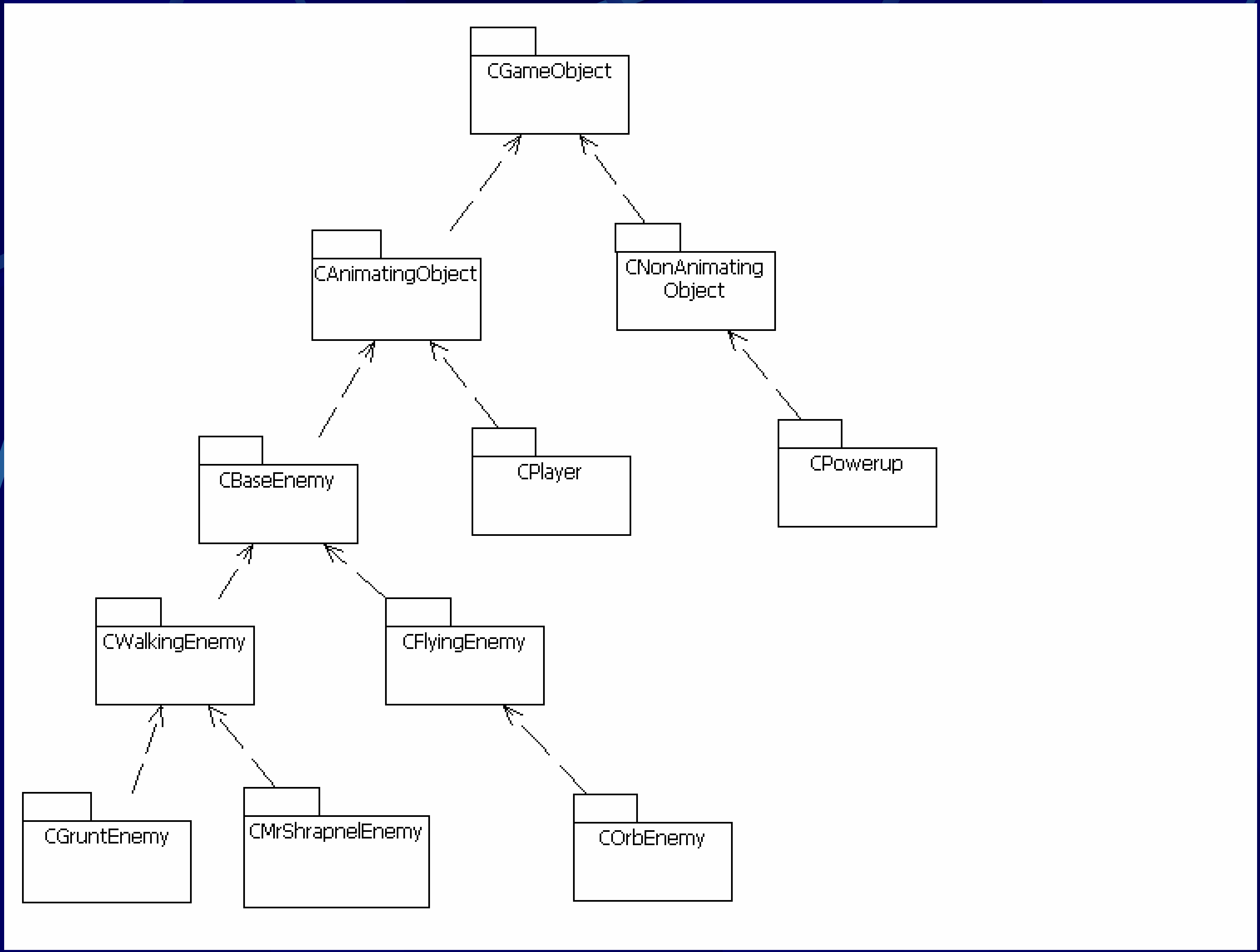
CGameObject
CAnimatingObject
CNonAnimating Object
?
CBaseEnemy
CPlayer
CPowerup
?
CAnimating Powerup
CWalkingEnemy
CFlyingEnemy
CGruntEnemy
CMrShrapnelEnemy
CBulletBill
COrbEnemy

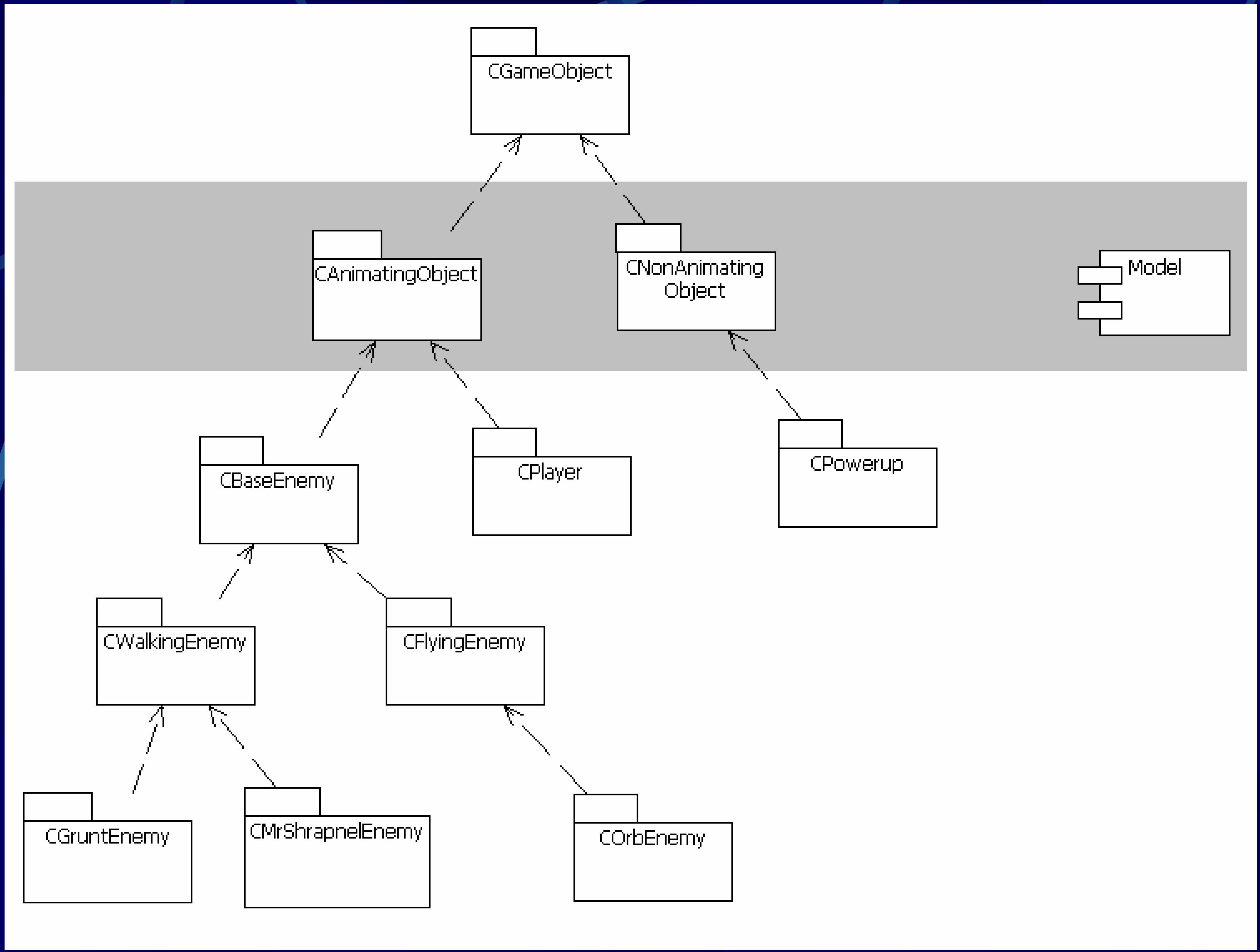
Inheritance hierarchy Summary

- The class-based hierarchy approach has serious problems with scale
- Very complex
- Not robust
- Poor extendibility

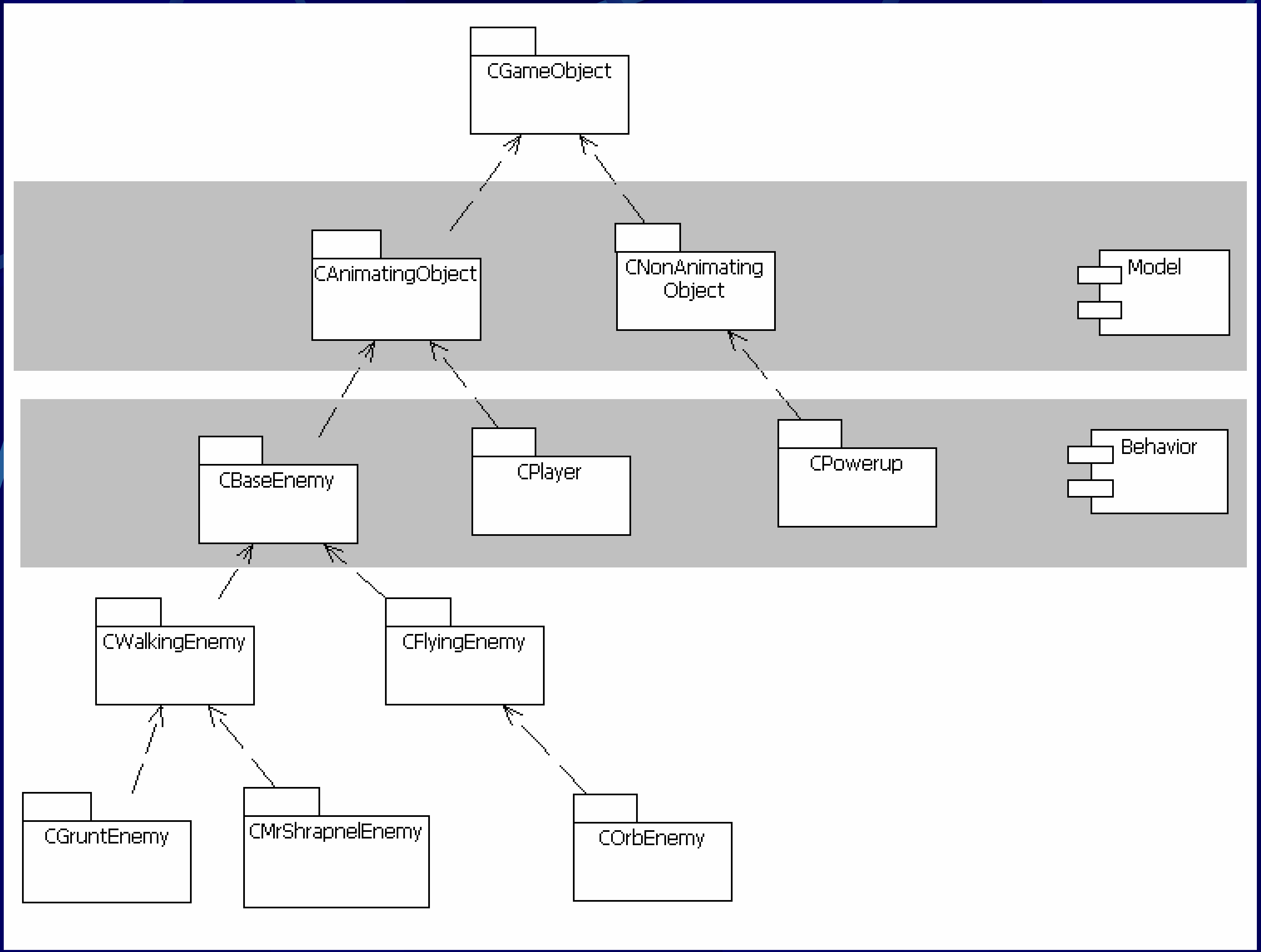
Component-based architecture

- Instead of inheritance, we use containment.
- Take each level of the class hierarchy, and change it to a component.
- This technique is called “Levelizing”.

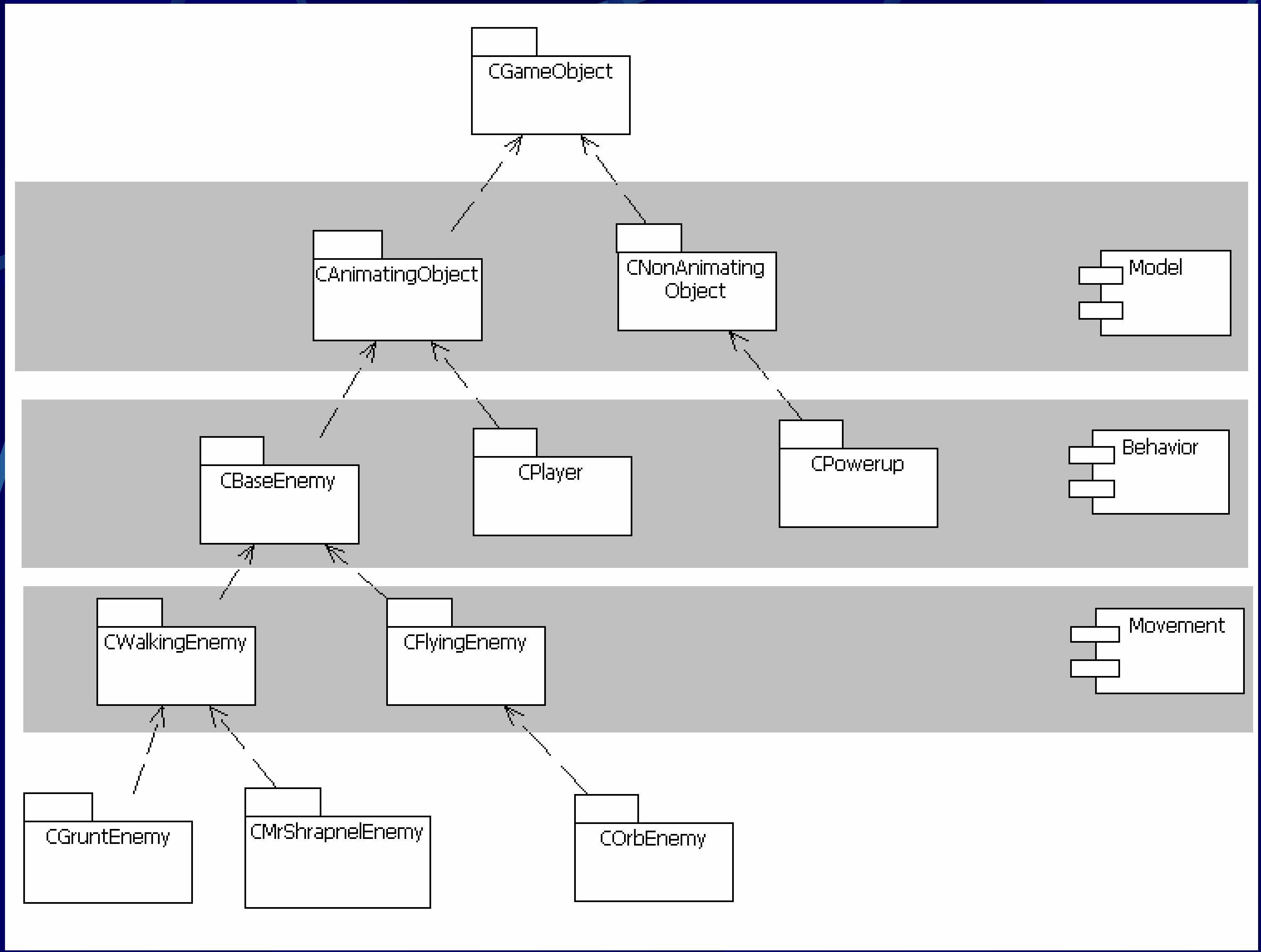

CGameObject
CAnimatingObject
CNonAnimating Object
CBaseEnemy
CPlayer
CPowerup
CWalkingEnemy
CFlyingEnemy
CGruntEnemy
CMrShrapnelEnemy
COrbEnemy



CGameObject
CAnimatingObject
CNonAnimating Object
Model
CBaseEnemy
CPlayer
CPowerup
CWalkingEnemy
CFlyingEnemy
CGruntEnemy
CMrShrapnelEnemy
COrbEnemy

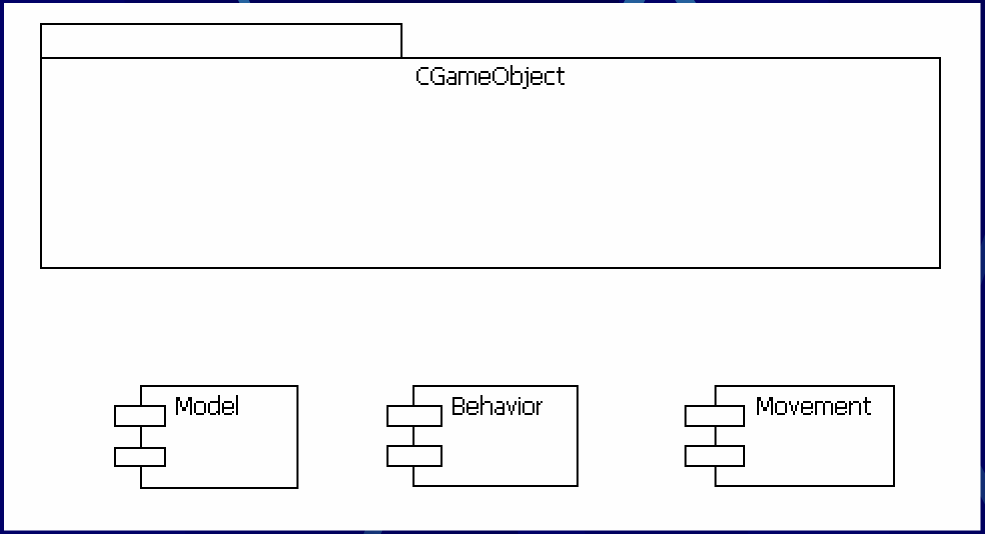


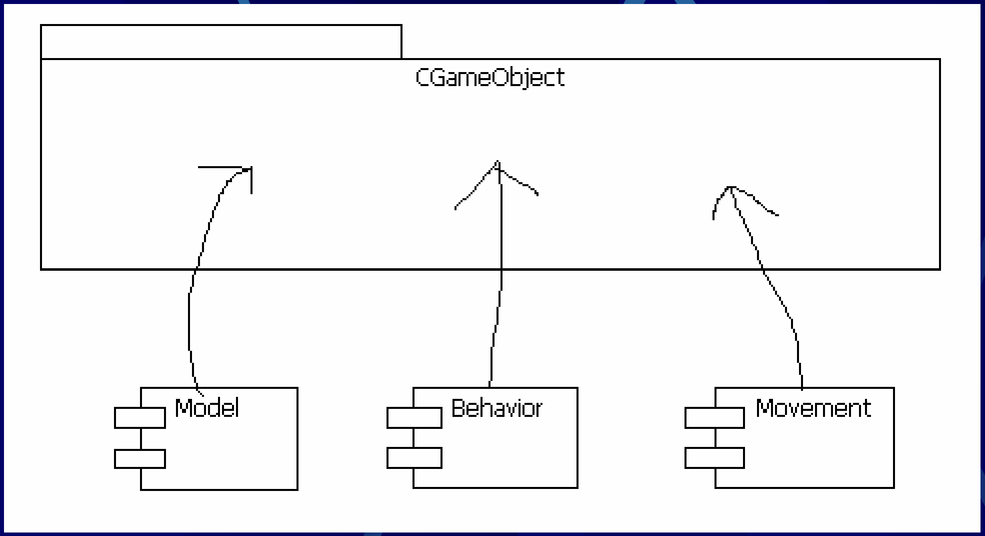
CGameObject
CAnimatingObject
CNonAnimating Object
Model
CBaseEnemy
CPlayer
CPowerup
Behavior
CWalkingEnemy
CFlyingEnemy
CGruntEnemy
CMrShrapnelEnemy
COrbEnemy

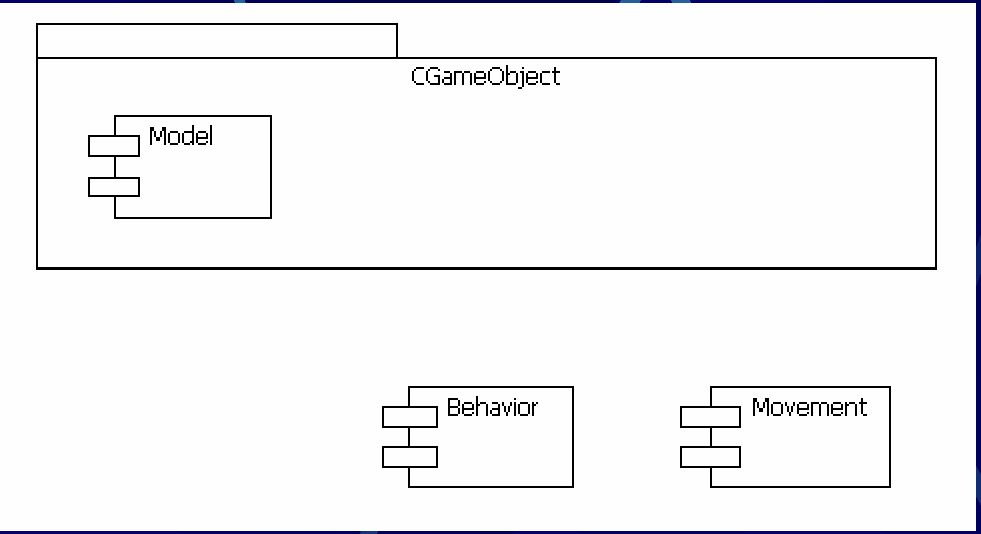

CGameObject
CAnimatingObject
CNonAnimating Object
Model
CBaseEnemy
CPlayer
CPowerup
Behavior
CWalkingEnemy
CFlyingEnemy
Movement
CGruntEnemy
CMrShrapnelEnemy
COrbEnemy

Levelizing, 2

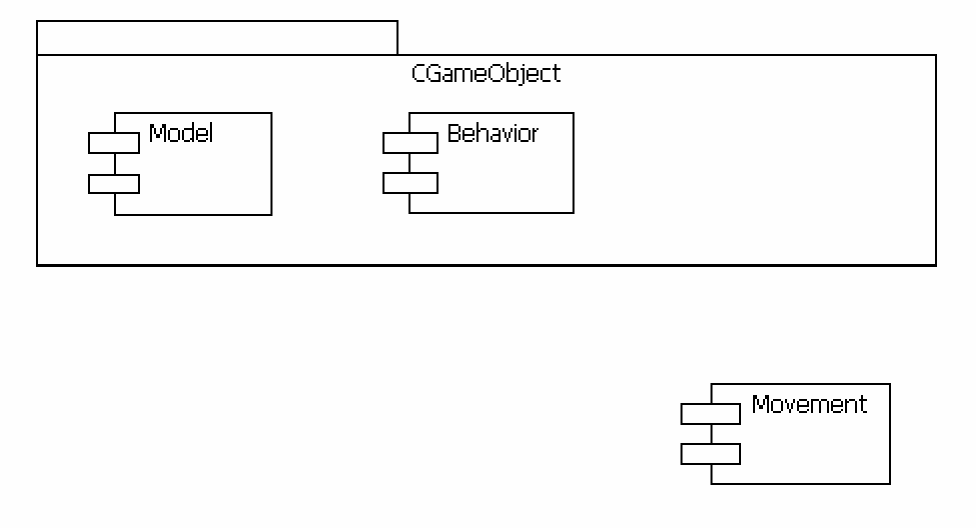
- Model
  - Animating
  - Non-animating
- Movement
  - Flying
  - Walking
- Behavior
  - Enemy
  - Player
  - Powerup

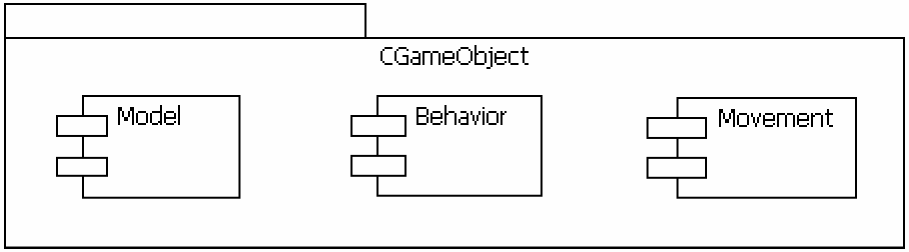

CGameObject
Model
Behavior
Movement


CGameObject
Model
Behavior
Movement



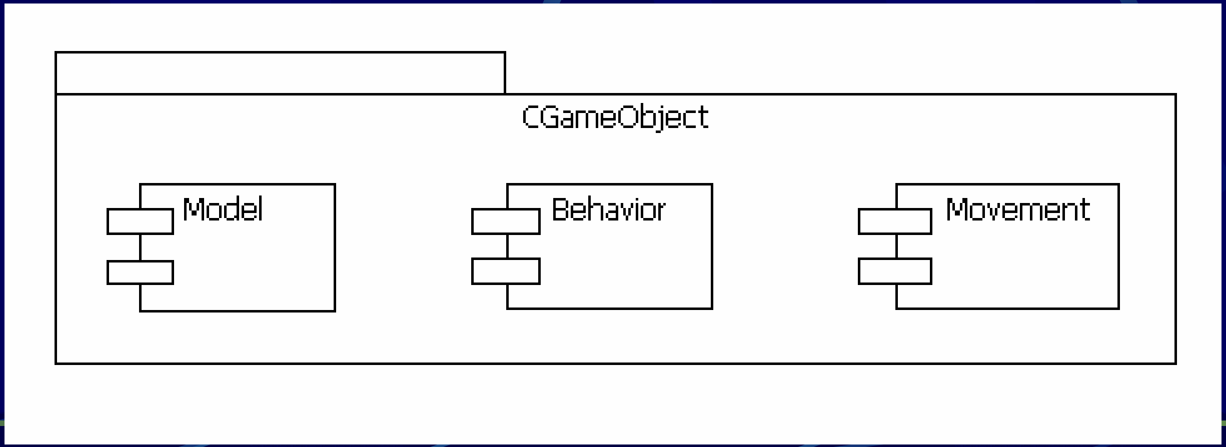
CGameObject
Model
Behavior
Movement


CGameObject
Model
Behavior
Movement


CGameObject
Model
Behavior
Movement

- If we apply this approach, we end up with the following for CGameObject:

```
class CGameObject
{
    CModel* m_model;
    CMovement* m_movement;
    CBehavior* m_behavior;
};
```

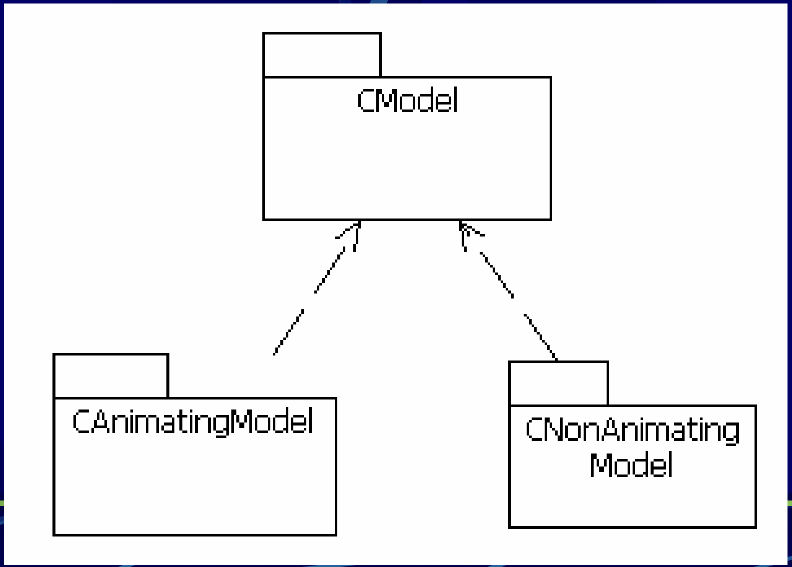


Implementing the components

- There are several methods for converting a level of the hierarchy over into a component
- The first method involves using C++ inheritance – make a base class that each class on that level can inherit from
- The second method involves containment – you put all of the functionality in the base component class

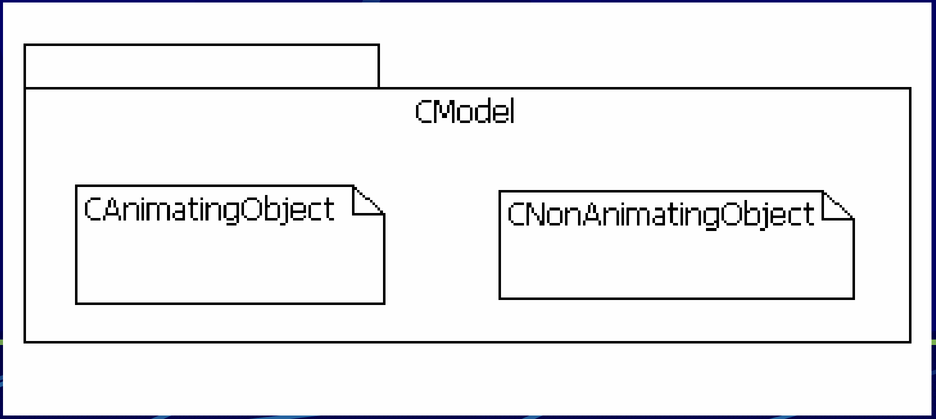
Inheritance Implementation

```
class CModel // the base class for model types.  Does not inherit from CGameObject
{
   // etc.
};


class CAnimatingModel : public CModel
{
   // etc.
};

class CNonAnimatingModel : public CModel
// etc.
```



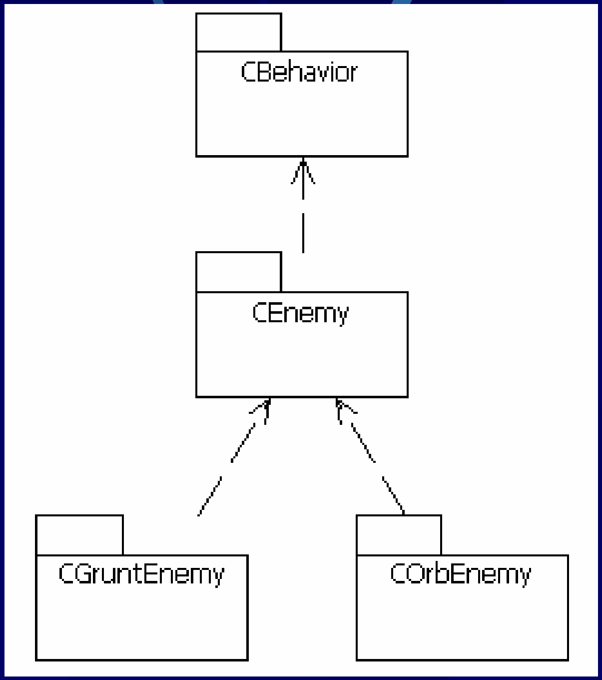
Containment Implementation

```
class CModel
{
    enum eModelType
    {
        MODEL_ANIMATING = 0,
        MODEL_NONANIMATING = 1,
    };

    eModelType m_model_type;

    union
    {
        animation_data* m_anim_data;
        static_data* m_static_data;
    };

    // etc.
};
```



Implementation details, 4

- Sometimes you'll actually end up with a class inheritance hierarchy off one of the components



Implementation Details, 5

- Class hierarchies aren't completely evil
- They just have serious scale problems
- For very simple hierarchies, they work fine
- Good rule of thumb: Don't have a hierarchy with more than 3 levels
- If the hierarchy needs to get deeper, you should levelize it

Advantages of component-based architectures

- Provides code and interface re-use without scalability problems
- Works well with the systems approach
  - Physics system – movement
  - Rendering system – model
- Composition of objects isn’t fixed at compile time – can change while the game is running.

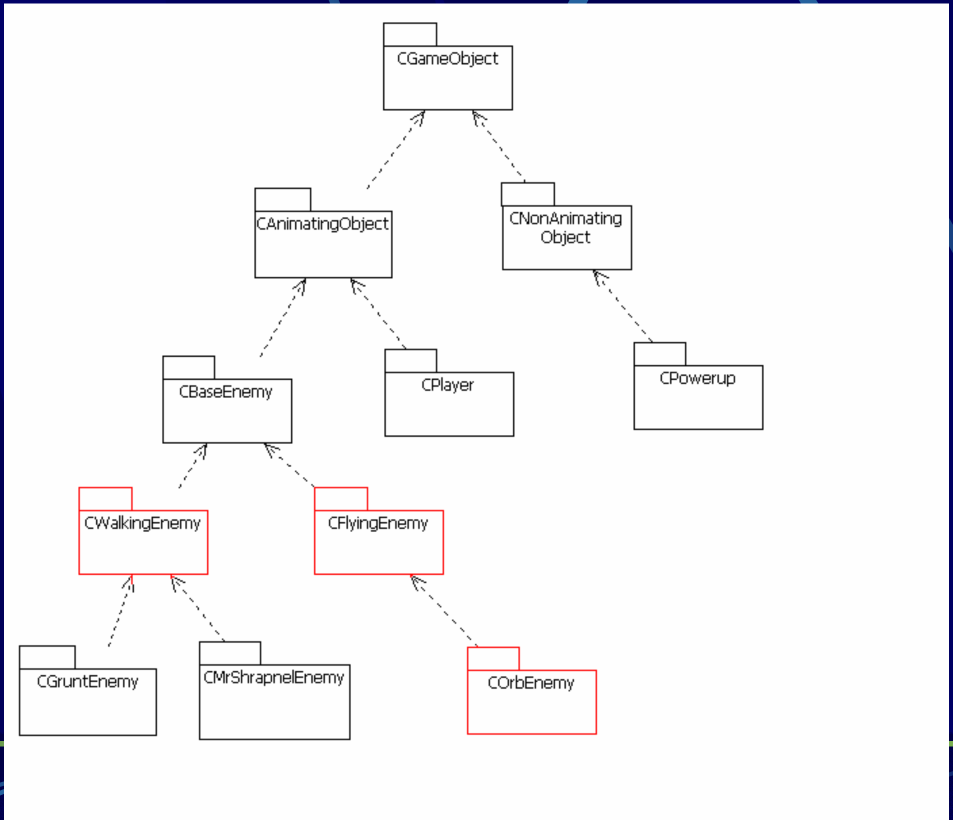
Token Architecture Summary

- Code and interface re-use are highly desirable for game object functionality
- Inheritance-based class hierarchies should only be used on limited numbers of object types
- Use a component-based architecture for anything significant
- Component-based architectures use containment rather than inheritance
- Component-based architectures also give run-time flexibility

Technique 4: Prototype-instance approach

Token Composition

- In a inheritance-based token hierarchy, the composition of each token is determined at compile time by the inheritance tree



Token composition, 2

- In the component token architecture, every token is composed out of a number of different components and sub-components
- The composition of a token is not defined at run time.

```
class CGameObject
{
    CModel* m_model;
    CMovement* m_movement;
    CBehavior* m_behavior;
};
```

Prototype definition

- Each kind of token in your game has a ‘prototype’ that describes the basic traits of the token
- Example - Smash TV 3D enemies:
  - Grunts
  - Orbs
  - Mr. Shrapnel
  - Tanks

Prototypes

- Prototypes specify the components that compose a token
- They can also specify the values of some of the variables in those components
- Prototypes are shared among tokens of the same type
- Example: Each different enemy type has a unique behavior, appearance, score, etc.
- This information is the same for all instances of that enemy type
- All of this shared information makes up the prototype

Instances

- Some information is instance specific - it can vary even among tokens of the same type
- Examples:
  - Location in the level
  - Velocity
  - Current hit points
  - Animation state

Protoypes and instances

- When an token is created, we want it to be initialized with the properties in its prototype
- We only need to store one set of data for each prototype
- Some data we want to store for every instance of the token that currently exists in the game world. This is called ‘instance-specific data’.

Prototype data – Where does it come from?

- In Smash TV 3D, we had various types of weapon projectiles in our games:
  - Rockets
  - Grenades
  - 3-way spread shot
- Each of these projectiles had corresponding prototype data
- The weapons were created through a factory function

Case study, part deux

- A common approach is to initialize a token's prototype data in the creation routines for the token
- You probably want to have a listing of all of the weapons side by side, to compare their properties
- In Smash TV 3D, we had a bunch of enums and #defines at the top of a file

Weapons example

```
enum
{
   kRegularShotDamage = 1,
   kRegularShotFireDelay = 200,

   kSpreadShotFireDelay = 200,
   kSpreadShotDamage = 1,

   kRocketShotFireDelay = 250,
   kRocketShotDamage = 1000,

   kGrenadeFireDelay = 50,
   kGrenadesDamage = 2,
};


static const char* kStrRegularShotMeshModelName = "bullet.model";
static const char* kStrSpreadShotMeshModelName = "3waybullet.model";
static const char* kStrShrapnelMeshModelName = "shrapnel.model";
static const char* kStrRocketMeshModelName = "rocket.model";
static const char* kStrTankShotMeshModelName = "evilshot.model";
static const char* kStrGrenadesMeshModelName = "grenadeshot.model";
```

Case Study, continued

- Whenever a weapon was constructed, the factory would initialize it's variables.

```
p_Weapon->SetDamage(kRocketWeaponDamage);
p_Weapon->SetSpeed(kRocketWeaponSpeed);
```

- This approach has many of the problems we discussed in the Pac-man case study.
- The variables are all in code, which is a bad place for variables that change frequently
- During development, a good deal of time was wasted tweaking variables and then recompiling

Improvements

- We want to move this information into a text file or database (the “data source”)
- This would allow us to just modify the data source to test your changes
- How do we associate the information in the data source with the game?

Prototype IDs

- A prototype ID corresponds to an entry in our data source that contains information for a specific prototype

Prototype ID	Name	Damage	Speed	Amount	Delay	Model
0	Normal Shot	1	10	42	200	Normal.model
1	Rocket	2	5	42	200	Rocket.model
2	SpreadShot	1	10	42	250	3WaySpreadShot.model
3	Grenades	1	5	84	50	Grenades.model

- This ID should be globally unique

Advantages of Prototype IDs

- We can avoid storing any of the prototype data in the class at all. Instead of:

```
m_damage = kRocketWeaponDamage
```

- We get:

```
CWeapon::GetDamage()
{
  return GetWeaponDamage(m_prototype_id);
}
```

Where `m_protoype_id` would be initialized to `PROTOTYPE_ROCKET`.

Prototype summary

- All of the information for your prototypes should be stored externally in a data source
- The game should read in the data source at run-time and keep a database of prototypes in memory, each entry storing all of the information for a specific prototype
- Each instance of a token should have a prototype ID so they can reference their prototype when it is needed

The Mr. Potatohead Prototype

Mix N' Match

- You don't need to have a different prototype for every token that describes ALL of the token's properties explicitly

Name	Model Type	Model Name	Movement Mode	Movement Speed	Movement Turn Rate	AI type	AI Aggressiveness	AI Morale	AI Learning
Grunt	ANIMATING	Grunt.model	Walking	5	360	ENEMY	0.5	10	No

- Instead, you can have prototypes corresponding to each component in the token
- Then a prototype for a token will actually be composed of sub-prototypes

Mix n' Match, continued

- In Smash TV 3D, we could make prototypes for the following categories:
  - Model

Prototype ID	Name	Model Type
0	Grunt.model	Animating
1	MrShrapnel.model	Animating
2	Invincibility.model	Static

  - Physics type (movement mode)

Prototype ID	Movement mode	Speed	Turn Rate
0	Ground	5	360
1	Air	3	180
2	Air	10	30

  - AI

Prototype ID	Aggressiveness	Morale	Learning
0	0.5	10	Yes
1	0.7	8	No

More Mix n' Match

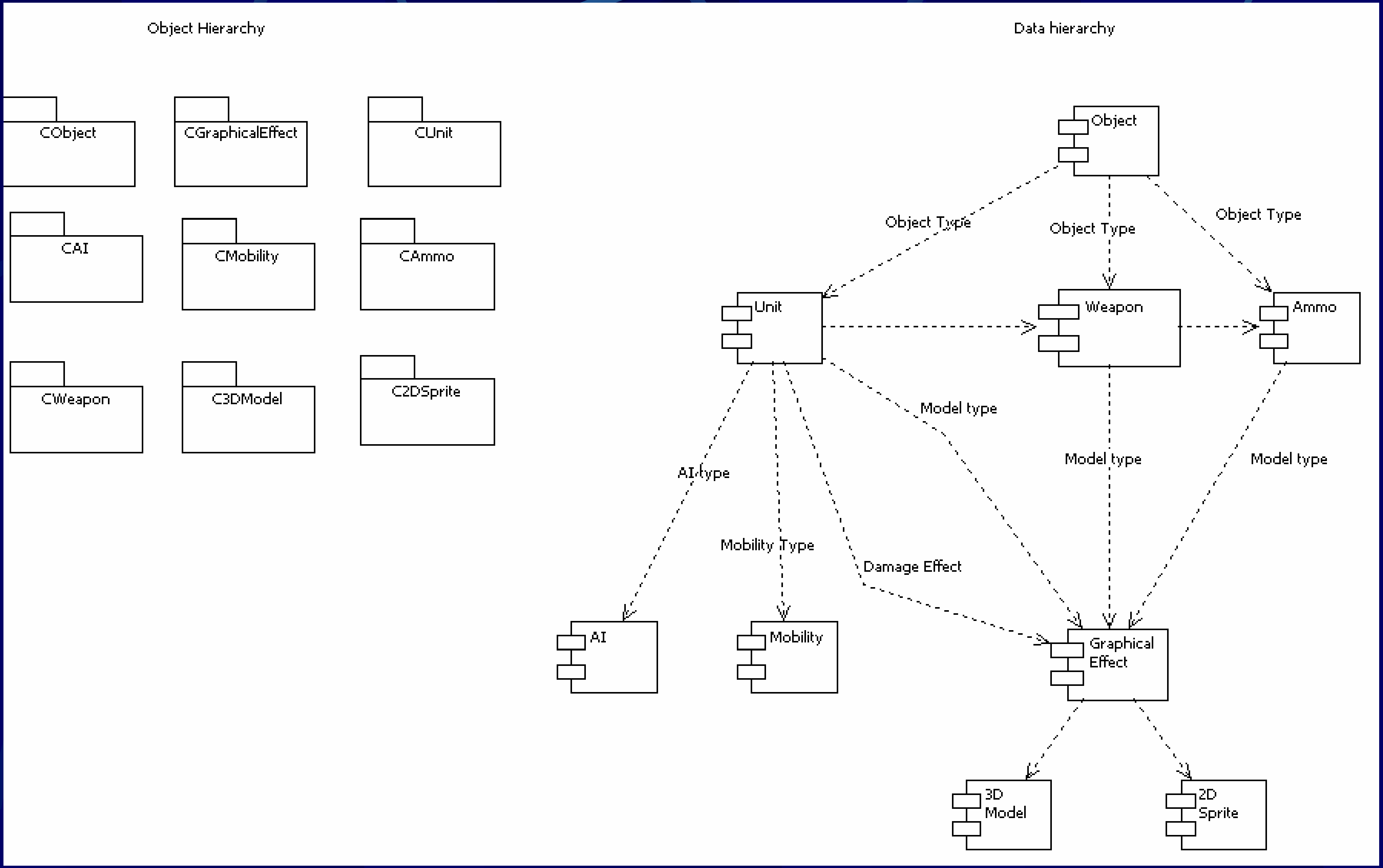
- You could then have a prototype for an enemy look as follows:
- Grunt Prototype:
  - Model: Prototype 0 (Grunt.model)
  - Physics: Prototype 0 (Walking)
  - AI: Prototype 1
- The Grunt token is then composed of several sub-prototypes, each which describes one of the grunt's components (model, physics, AI)
- This allows different tokens to share sub-prototypes

Extending Mix n' match

- Prototypes can also reference other prototypes
- Example: The grunt prototype could reference a specific weapon prototype, indicating the weapon the grunt will use – e.g. "Rocket Launcher"
- The rocket launcher might in turn reference a 3d model indicating the visible appearance of the weapon when wielded
- The player could also reference the same weapon prototype
- Using this scheme, the code could treat the player's weapon and the enemy's weapon identically
- This buys consistency and reliability because we get code and interface re-use in-game

Mix n’ match

- Sub-prototypes can be further split into sub-sub prototypes
  - Ground movement prototype could be split into ‘treads’ and ‘walking’, etc.
  - This process can continue ad infinitum
- Using prototypes with a component-based architecture results in a flat object hierarchy, but a complex data hierarchy



Object Hierarchy
Data hierarchy
CObject
CGraphicalEffect
CUnit
CAI
CMobility
CAmmo
CWeapon
C3DModel
C2DSprite
Object
Object Type
Object Type
Object Type
Unit
Weapon
Ammo
Model type
Model type
Model type
AI type
Mobility Type
Damage Effect
AI
Mobility
Graphical Effect
3D Model
2D Sprite

What about Instances?

- Once you have your prototypes, you want to have a way to create instances of them in your game and set up the instance specific data
- In Smash TV 3D, different types of tokens were handled differently
  - Enemies was created through spawn points. A text file controlled the timing of enemy creation.
  - The Spawn Points themselves were placed in the level in 3DS Max and were created when the level started in the appropriate positions
  - Powerups were created randomly every couple of seconds, based on a percentage in a text file

Instances

- Instances can be created through designer intervention and/or through systems
- The creation must provide all of the instance specific data that the token requires
- This is usually at least a position in the game world
- In Smash TV 3D, the spawn points gave each object its initial position (at the spawn point)
- Initial velocity was always towards the center of the room
- Powerups were created in a random position within a certain radius of the room's center

Factory Functions

- A common way to implement instantiation of tokens is through factory functions
- Factory functions take a prototype ID, and any instance-specific data that cannot be filled out by the data source, and create an instance of an object
- Usually look something like:

```
Some_Type_Of_Object_Handle_Or_Pointer
CreateObject(int PrototypeID, instance_specific_initial_params...)
```

- Factory functions insulate concrete entity details from systems that use and spawn those entities

Prototype-instance Summary

- Prototypes consist of the information necessary to describe the basic traits of a specific type of token in the game
- The instances of a prototype are the actual tokens of that type that exist in the game world
- Instances contain both a reference to their prototype (ID) and instance-specific data
- Game behavior is generated by tokens, their prototypes, and the systems that operate on those prototypes

Technique IV: Scripting

- The systems approach / component architecture tends to break down when the number of unique prototypes of a given type beings to grow dramatically, and when each prototype requires very different behavior
- Examples:
  - Scripted Events
  - Object AI implementation

Scripted Events / In-game Cutscenes

- You want a specific sequence of events to occurred in a specific timed order
- Each scripted event is unique
- Having a system to manage a bunch of unique events doesn’t make sense, since you just end up with a lot of very special case functions in code, one for each scripted event:

```
void PerformCutsceneWherePlayersEntireVillageIsWipedOut();
void PerformCutsceneWherePlayerFlashbacksToPresent();
void PerformCutsceneWhereShadowyFiguresDiscussPlansOfWorldDomination();
```

- This makes it difficult to tweak these sequences

That bothersome AI

- Object AI is similar in this regard
- Each AI is unique and provides a different set of behaviors
- Many of the behavioral variables need to be greatly tweaked before they look appropriate
- There is not enough in common among disparate AIs to move all of their variables to a data source without enormous amounts of effort

The general problem

- You will run into this problem whenever you’re dealing with some component that requires a large number of very different pieces of special case code
- The key word is ‘different’
- If you have a hundred different objects, but you can break them down into a few variables and behaviors, you can still use the system-prototype approach
- Each creature AI and cutscene has a vastly different behavior, even in a simple game

Grunt Variables

```
static const float32 kEnemyGruntBackswingSpeed = 5.0f;
static const float32 kEnemyGruntSwingSpeed = 15.0f;
static const float32 kEnemyGruntRecoverSpeed = 2.0f;
static const float32 kEnemyGruntSpeed = 15.0f;
static const uint32  kEnemyGruntNumSwings = 3;
static const uint32  kEnemyGruntBackswingTime = 300;
static const uint32  kEnemyGruntSwingTime = 100;
static const uint32  kEnemyGruntRecoveringTime = 500;
static const uint32  kEnemyGruntRecoveredTime = 200;
```

Rhino Boss Variables

```
static const uint32  kBossRhinoEntranceDuration = 3000;
static const uint32  kBossRhinoTurnTime = 2000;
static const uint32  kBossRhinoDeathTime = 300;
static const uint32  kBossRhinoTimeBetweenAttacks = 5000;
static const uint32  kBossRhinoTimeBetweenShots = 100;
static const uint32  kBossRhinoPrepShotTime = 1000;
static const uint32  kBossRhinoPrepChargeTime = 3000;
static const float32 kBossRhinoChargeAdjust = 0.95f;
static const uint32  kBossRhinoChargeAdjustTime = 250;
```

and only a few variables in common:

```
static const uint32  kBossRhinoHealth = 500;
static const uint32  kEnemyGruntHealth = 1;
```

What do we want?

- Each custom behavior should be able to function in isolation from other custom behaviors, so it is easier to test and manage
- We want to be able to tweak as easily as possible
- Behavior should be determined at run-time, rather than compile-time, for the above reason
- Behavior should be treated as “data”, rather than code, and be managed accordingly

Scripting Languages

- Scripting languages are programming language other than the one actually used to program your game, and are used them to write game behavior through an API
- You pass the scripting language through tools that compile the language into a format that can be loaded on the fly and executed

Problems with Scripting Languages

- Writing one is not for the faint of heart
- Require an enormous amount of work
- You have to write a compiler and executer
- You have to expose an API to them
- You will have to write a parameter marshaller
- You will want a nice suite of tools, like a development environment, debugger, etc.
- There will be learning curve since it's a brand new language
- May result in unmaintainable spaghetti code if poorly designed
- Usually at best 1/10 speed of C/C++, so can drag down performance
- Existing scripting languages (Python, Ruby) may not meet your needs, there's still a learning curve, and will still run very slowly

Dlls

- All we really want is to be able to write in code, and have that code loaded at run-time rather than linked in at compile-time
- On Windows, DLLs do exactly that
- Very similar to executables – collections of methods and variables in a specific format
- DLLs cannot be launched directly
- Another executable or DLL must load the DLL into its memory space, and then it can use the DLLs functions

Using DLLs

- You should encapsule each different behavior in its own DLL
- Within each DLL, you'll probably need to expose a few entry points
  - A Run() method
  - An Initialize() method
  - A Shutdown() method
  - A way to notify the script of events
- Keep entry points to a minimum; your game should communicate with the script in your DLL through a limited number of connections

Using DLLs, 2

- You will need to provide an API to allow the script to communicate with the rest of the game
- Approach 1:

Write another DLL that contains the API. The game should load this DLL on startup and intialize it with access to all of the appropriate global variables. These variables should be stored in a shared segment of the DLL. All script DLLs should link against the LIB provided with this DLL

- Approach 2:

When the script starts up, pass it a structure that contains a bunch of function pointers to all of the API functions (or optionally a class with pure virtual functions)

Using C++ classes in scripts

- Because of name-mangling, you can't place the exported functions in a class
- To avoid this problem, you can use a wrapper approach. In your initialize(), allocate an instance of the class you want to use as a global variable. Then, have the Run(), CallEvent(), and other exported functions just forward the call to functions in the instantiated class.

A better approach

- Create an abstract base class with the functions that you would have normally exposed in your DLL.

```
class CScriptDLL
{
public:
   virtual bool Initialize() = 0;
   virtual bool Shutdown() = 0;
   virtual bool Run() = 0;
};
```

- You cannot instantiate CScriptDLL.
- You have to create a class that inherits from it, and implement all its the pure virtual functions

```
class CCreatureAI: public CScriptDLL
{
public:
   virtual bool Initialize() {...}
   virtual bool Shutdown() {...}
   virtual bool Run() {...}
};
```

- Now, any code that knows about CScriptDLL can call these functions through a CScriptDLL pointer, even if it actually points to a CCreatureAI.
- Now, instead of exposing all of these functions in the DLL, you only need to expose one function:

```
CScriptDLL* EXPORT CreateDLLInstance()
{
   return new CCreatureAI;
}
```

- When the executable loads your DLL, it only needs to call CreateDLLInstance(), hold on to the returned pointer, and use it to call the Initialize(), Shutdown(), and Run() functions

Scripting Languages vs. DLLS

- There are tradeoffs between both scripting languages and DLLs
- The argument that scripting languages are easier to use for non-programmers is irrelevant; most complex behavior can only be written by someone could also understand C/C++
- One advantage scripting languages can offer is language features difficult or impossible to realize in C/ C++
- For DLLs, you must have a version of Visual C++ or the equivalent on the machines that will be compiling scripts
- For a game like Quake, where a lively mod community is desired, this may not be an option
- On consoles, you will have to learn the executable format and write your own linker to use the DLL method
- This is still easier than writing a scripting language

Technique VI: The Data Editor

- The prior techniques have suggested ways to move much of the game behavior into a ‘data source’ external to the code
- We want to be able to edit the ‘data source’ as easily as possible
- A database can be a good choice for your data source, since they are easy to read and edit

The Data Editor, 2

- Databases are only good for numerical and text data
- For graphical data, you should write new or leverage existing tools to manipulate the data in a graphical fashion
- Try to use existing tools when possible
- In Smash TV 3D, we used 3DS Max to create levels and place spawn points, collision data, and doors
- At Outrage, we use Maya for editing of objects and levels, and have written a whole suite of plugin tools to add game content
- Even for text and numerical data, it can also be useful to write dialogs that wrap the database for ease of editing

Technique VII: ASCII vs. Binary

- If you have a lot of files that get loaded into the game, you can adopt a dual ASCII / binary approach
- ASCII is easier to debug and validate
- Binary is faster and reduces file size
- You can have all your tools write out ASCII files in a manner that can be easily read in
- Also provide a tool that convert the ASCII into tightly packed binary
- If you add binary / ASCII transparency into your file I/O libraries, you will get the best of both worlds
- Don’t read in the database as is either; have tools to convert it to an easily-read binary form

Technique VIII: Reload game content on the fly

- Editing values in the database does not require a recompile
- It still requires a restart of the game and a reload
- A system that lets you reload portions of the game data while the game is actually running can save huge amounts of time
- You can even add a button on the data editor to automatically signal the game to reload its game data, making the process automatic

Technique IX: Debug global variables

- DLLs / scripts often have dozens of variables that need to be tweaked
- We must recompile or at least reload the script to make any changes
- This slows down tweaking of these variables
- It is also far too time-consuming to expose all of the variables in DLLs / scripts individually in the data source
- Instead, create some generic debug global variables (some floats, some ints, etc.)
- Use the debug variable in place of the real variable while testing values that need tweaking
- Since they are global, you can change their values in the debugger
- This will give you instant feedback for tweaking

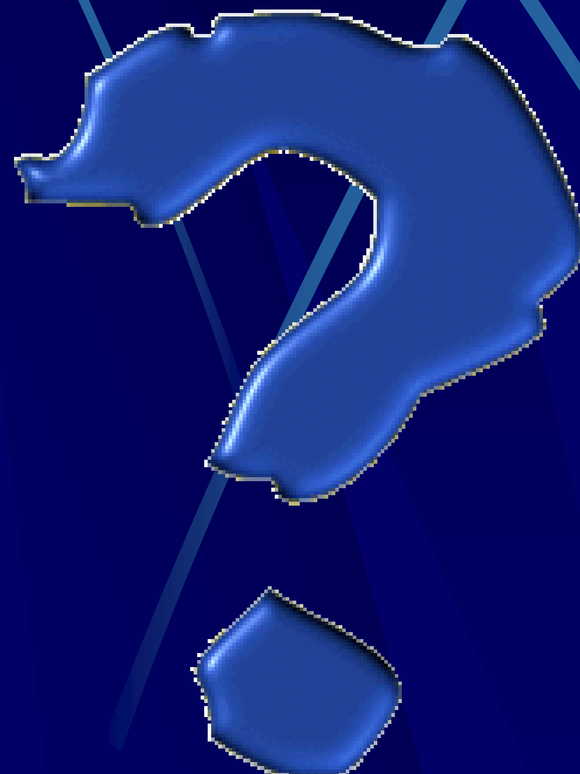
Caveats

- Don’t lose sight of your original goal
- All of these techniques require additional work up front before you see any results
- Take the timeframe and design of your project into account
- Don’t write code that won’t be used immediately
- The best way to design for the future is to have a solid design and clean, well-commented code

Data-driven design Summary

- Moving game behavior out of code and into data results in a more efficient process for tweaking content and gameplay
- Use a component-based architecture with the prototype-instance approach for easily tweakable and incredibly flexible game tokens
- Structure your game systems to operate on specific tokens and / or token components

Questions?



THE END

Techniques and Strategies for Data-Driven Design in Game Development

Scott Shumaker

Programmer,

Outrage Games

sshumaker@outrage.com