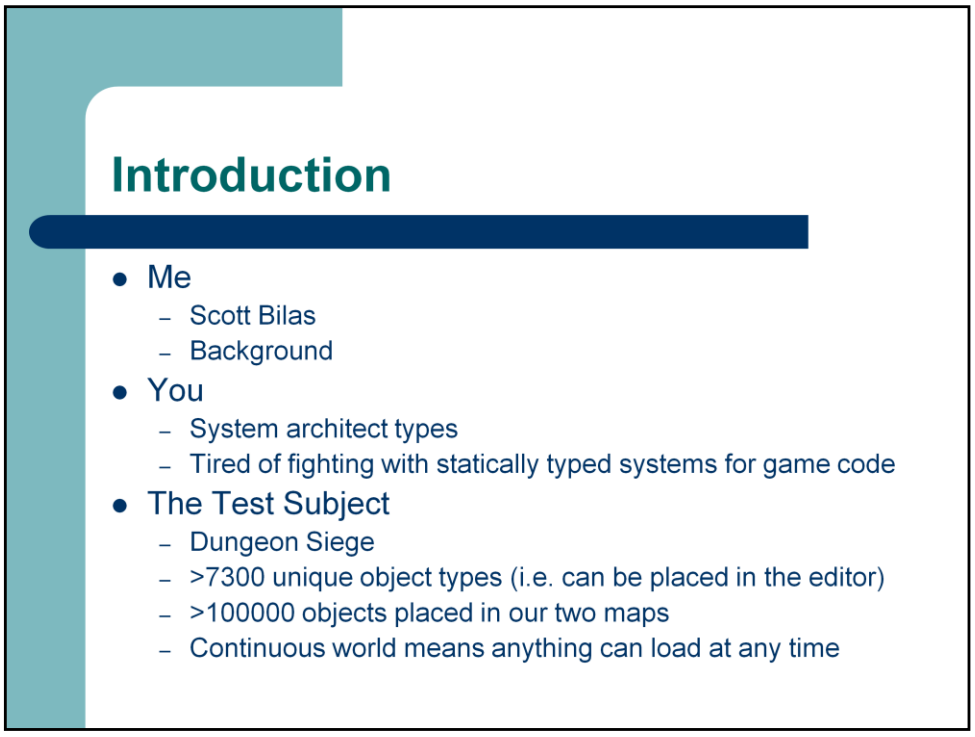
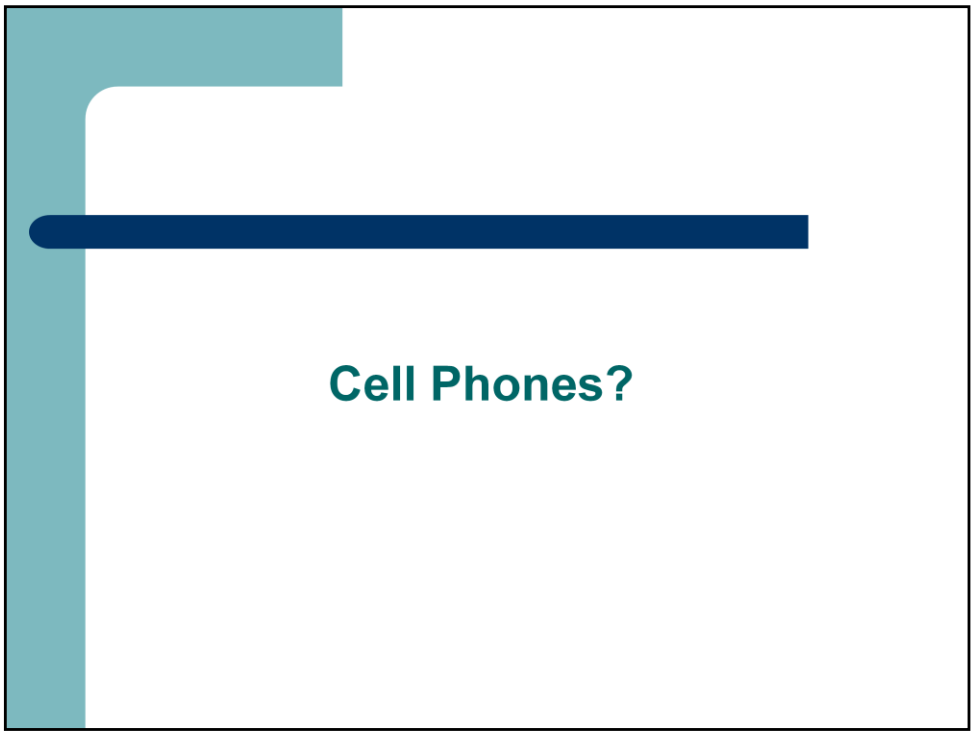
A Data-Driven Game Object System

GDC 2002

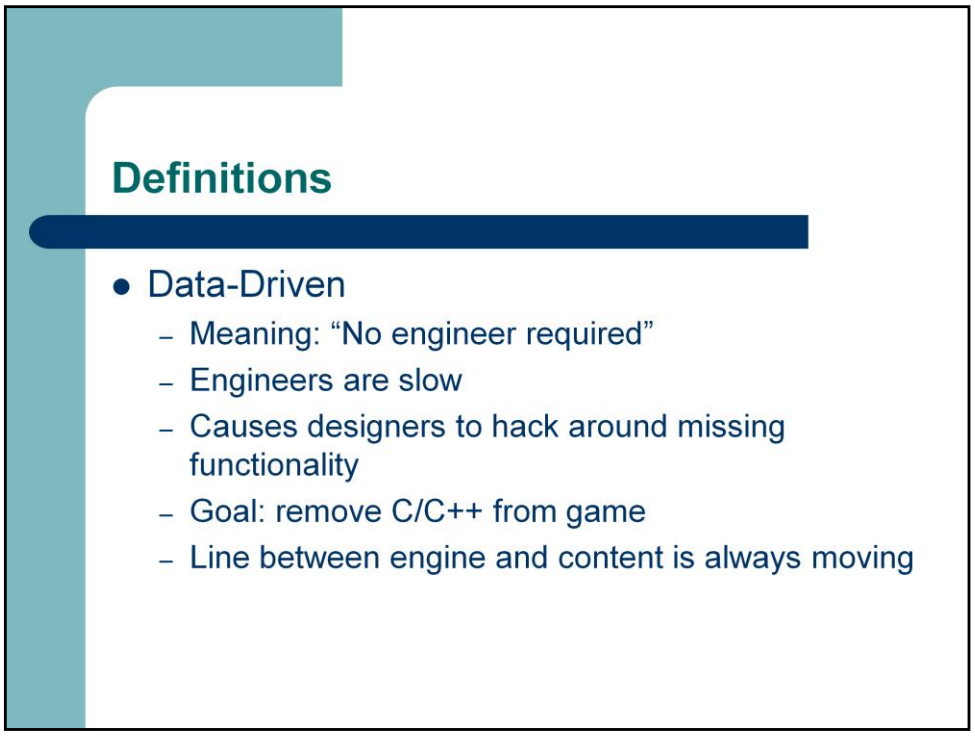
Scott Bilas
Gas Powered Games



I implemented the system I'm about to describe about a year and a half ago. As with most things in a game engine, it ended up getting used in many ways I never imagined. This talk is about the problem I was trying to solve, what I did, why you should care, and how it all turned out.



Oh wait, before I forget, please put away annoying little electronic beeping things that play TV theme songs and all that…

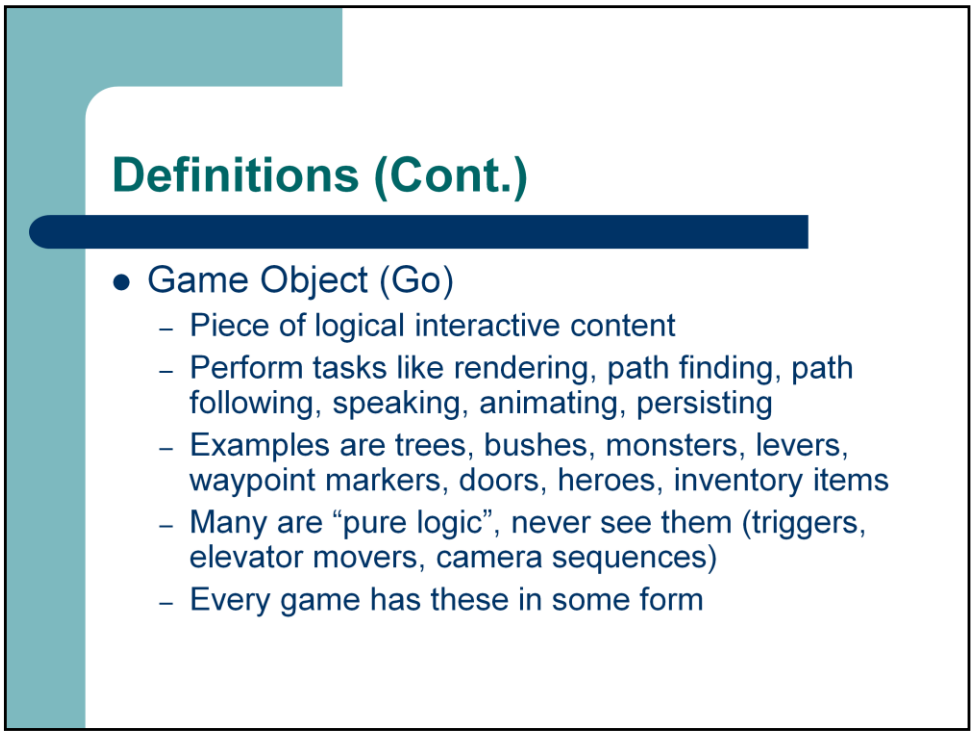


First let’s look closer at the title I chose for this talk and get some terminology out of the way.

Data-Driven

This can mean many things, but for me it means that you don’t need to involve an engineer when you want to change something. Engineers are really slow and it often seems like they take forever to get anything done. This of course drives designers and artists crazy. A non-data-driven system would have the names of animations to run for a monster hard-coded into the C++ files, requiring an engineer’s time and a new build to change. A data-driven system would store all of this in a data file somewhere and you could even write a tool to keep it updated.

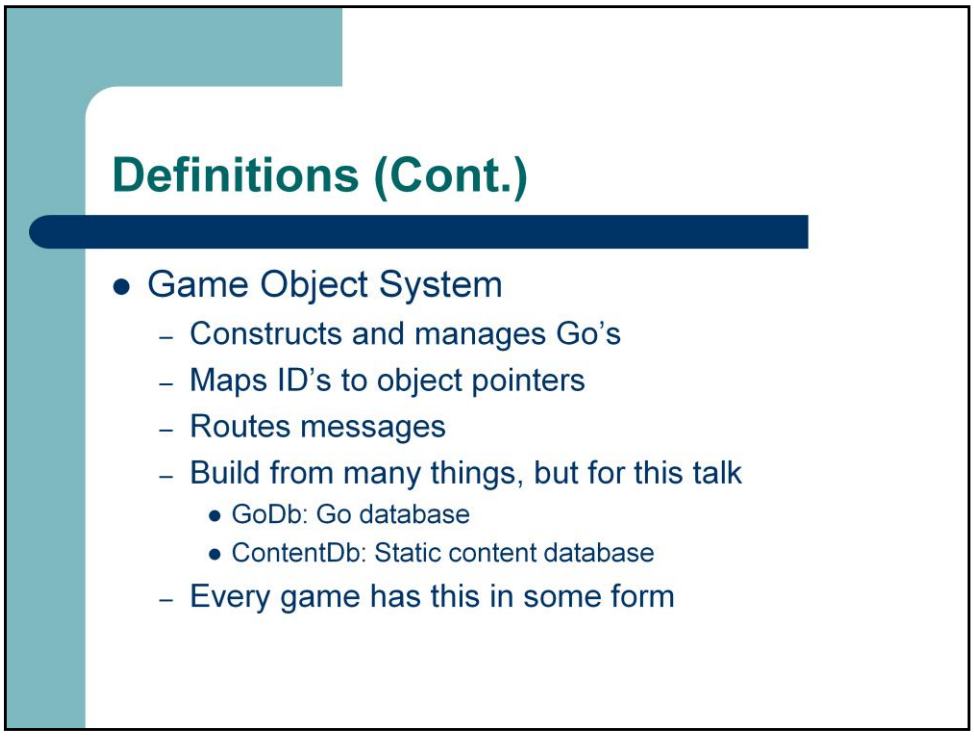
When I wrote the above, I realized that I was thinking in 1995 terms. Today we’ve gone much further than that, to the point that the code doesn’t even know what a monster is, and the only reason it knows to run animations is because some AI script with its own agenda is sending a request to do so. In many modern game engines you can make radical changes to the system without ever speaking to an engineer. This permits rapid prototyping and is a wonderful thing, if managed properly. The line between engine and content is always moving, maybe one day we’ll engineer ourselves out of existence.



Game Object

A Game Object or what we at GPG call a ‘Go’ is a piece of logical interactive content that the player can do something with. The line is blurred of course depending on the engine. In Dungeon Siege, some examples of Go’s are trees, bushes, monsters, levers, waypoint markers, and doors. The heroes and each item in their inventory (like swords, rings, armor, and the inevitable potions) are Go’s. Many Go’s you never see, such as triggers, elevator movers, sound emitters, etc.

Go’s are self-contained logic that can perform many tasks, or you might say they have many abilities. They might render themselves, find paths, follow paths, think for a bit, say something, ready and shoot an arrow (which is itself a Go), or self-destruct, spawning an effect on the way out. There’s nothing special about Go’s, every game has something like this, of course, but each game does it differently.

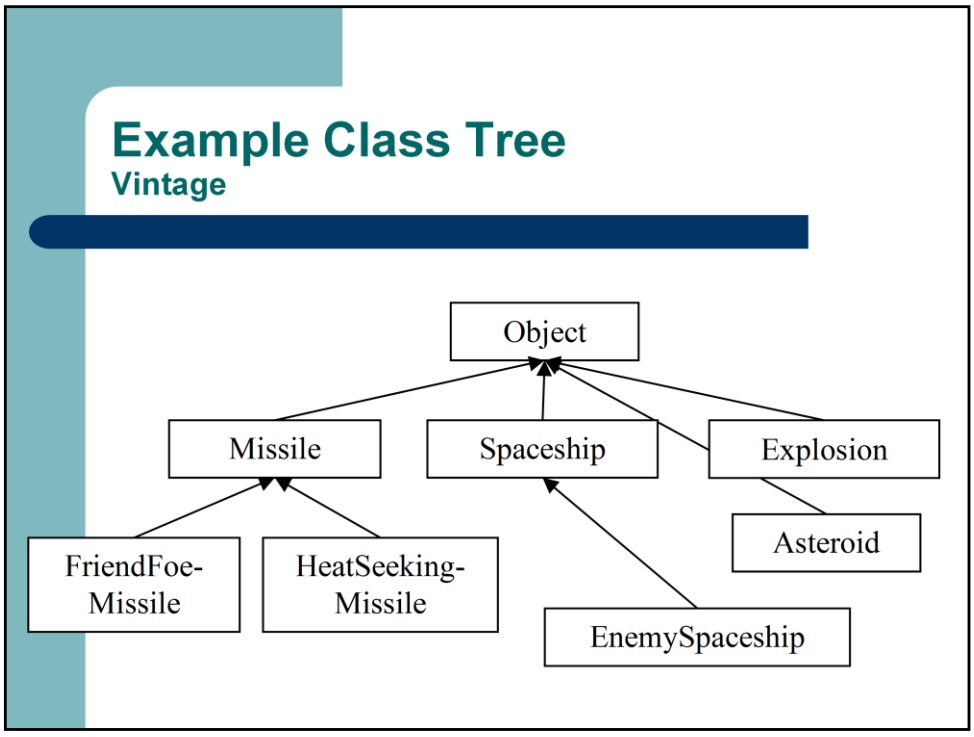


The Game Object System is simply the system that constructs and manages Go's for the game. It's responsible for mapping ID's to object pointers, creating and destroying Go's, managing external requests to use Go's, and routing messages. In Dungeon Siege this is built from a lot of systems, but for the purposes of this talk we'll be covering the GoDb (Go database) and the ContentDb (static content database).

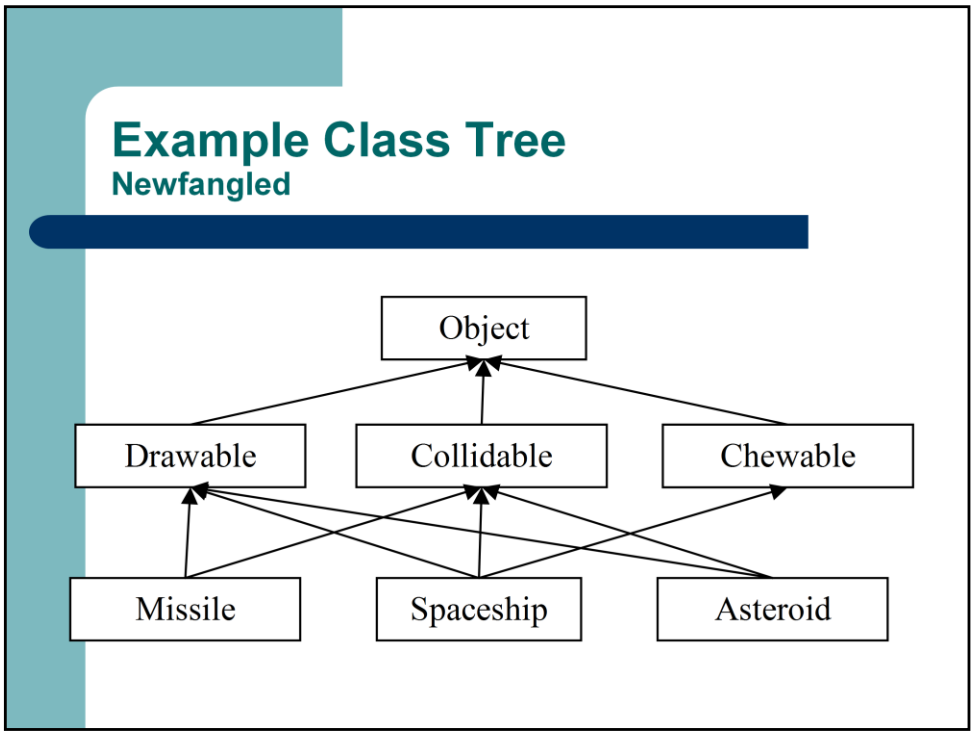
Games today have crazy amounts of content, and I'm not talking about movies, which are easy (i.e. each movie is a single piece of content, no problem!). And it's only getting worse (or better, depending on if you're a glass-is-half-full type of person).

So let's get into the main topic here. Say you're an engineer set out to create a new Game Object System from scratch, and you're going to "do it right the first time". You talk to your designer and say "what kind of content are we going to have in this game?" They respond with "oh lots of stuff, trees, and birds, and bushes, and keys and locks and …<trailing off>" and your eyes glaze over as you start thinking of fancy C++ ways to solve the problem. The object oriented programming sages tell you to try to determine Is-A relationships and abstract

functionality and all that other fun stuff. You go to the book store and buy a C++ book just to be sure, and it tells you to fire up your $5000 UML editor with one of the classic examples (next slide):

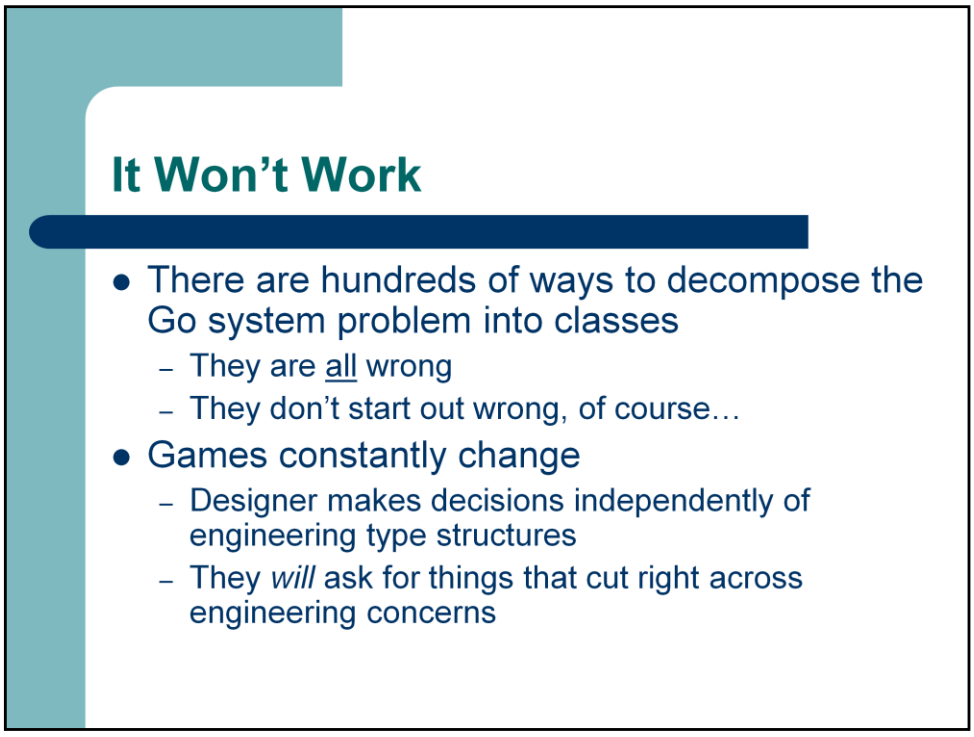


Here we have all our game types specified as classes in a nice hierarchy. I had to keep this diagram simple to fit on this slide but imagine all the fun virtual functions you'd see here, like DrawSelf(), Think(), and HandleMessage().



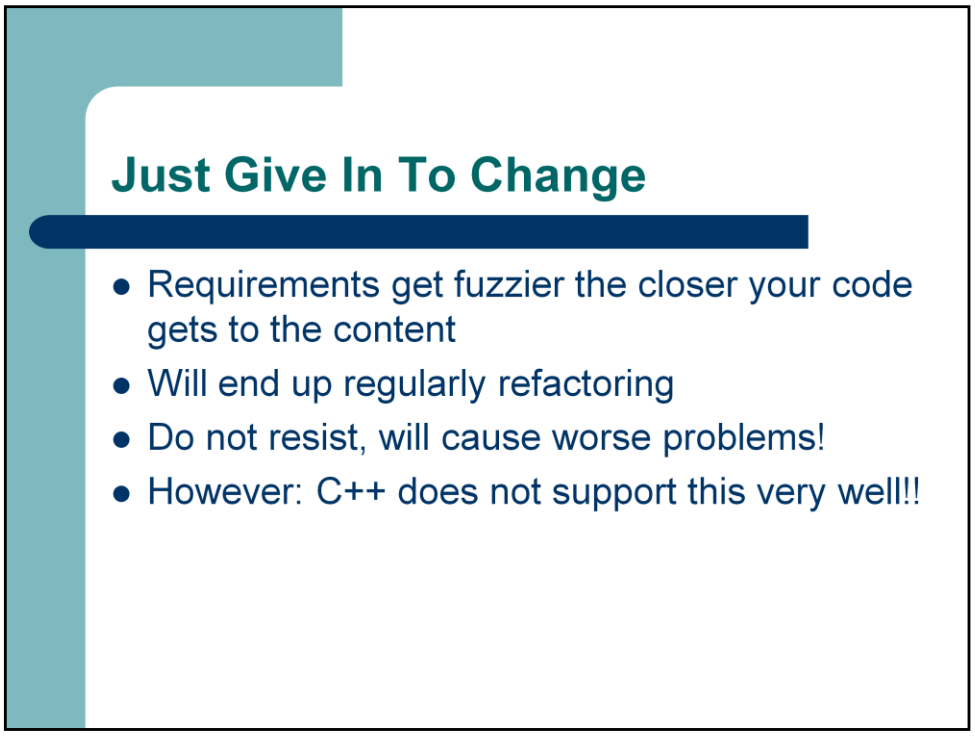
Fancier books recommend something like this.

Here, we try to decompose functionality along capability lines. Each “mixin” class adds functionality, for example the “chewable” class would add knowledge of how a space monster would be able to chew up the object (maybe play some effects, add info to the leaderboard, etc.). Again, no space on the slide, but expect that there would be hooks (pure virtual functions) like OnDraw(), OnGetBoundingBox(), and whatever else.

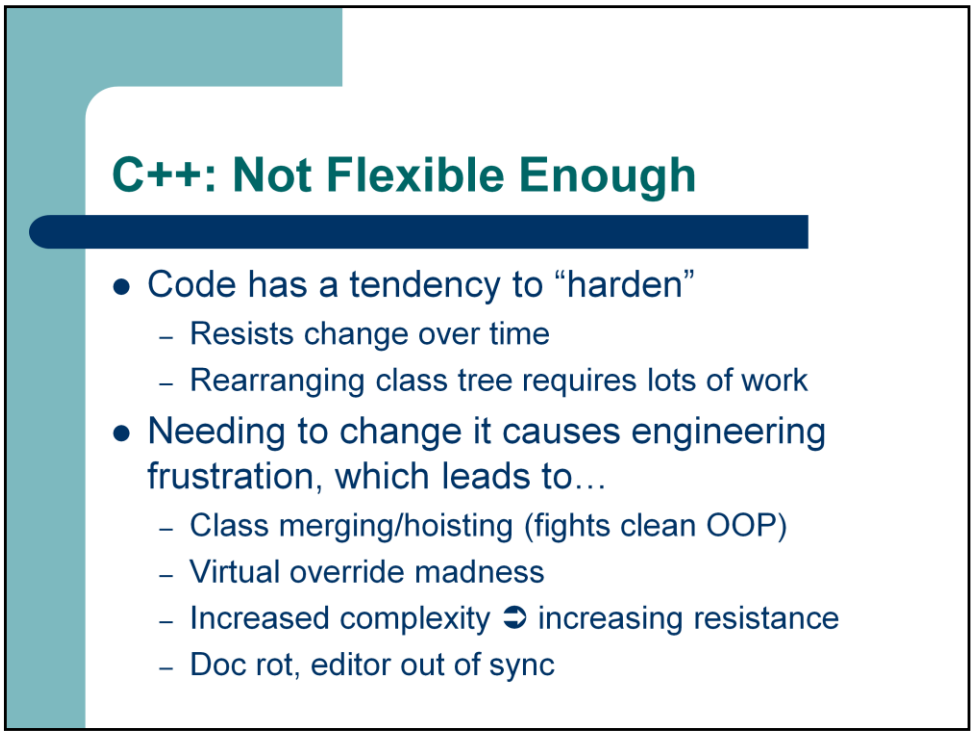


There are probably hundreds of ways you could decompose your systems and come up with a set of classes (I just showed a couple simple ones), and eventually, *all* of them are wrong. This isn't to say that they won't work, but games are constantly changing, constantly invalidating your carefully planned designs. How many post-mortems have you read about designs that were too ambitious, or massive changes mid-stream, or when marketing demanded some silly thing, or something needed to get added because somebody thought it might make the game more competitive...

So you hand off your new Game Object System and go work on other things. Then one day your designer says that they want a new type of "alien" asteroid that acts just like a heat seeking missile, except it's still an asteroid. Or they want to get rid of this whole spaceship concept and go underwater instead. Or they want those trees to sway back and forth in the wind…



The closer your code gets to that line between engine and content, the fuzzier the requirements and the more likely that your work will regularly need to get refactored. You could resist change like this, but that will just result in designers hacking around things, creating even worse problems. Suffice it to say that it's just easier to try to handle this change as a fact of normal game development. And how does that change get handled? In software engineering we learn to take the things that vary and abstract them. In other words, find the parts of the system that are likely to change and make them flexible. Traditional wisdom says to do this through Is-A relationships via the class tree. Unfortunately, the class tree resists change.



This can go wrong a number of ways, and all of them are caused by programmer frustration. Here are a few:

Class merging (sometimes called hoisting)

Over time, many of these little classes end up getting merged into larger monolithic classes. It’s just easier to have little bools inside the base class that say “turn this feature on”. Many times the bools are “turn this feature off”, because you figure out that 90% of your derived classes are duplicating the same code, and it’s easier to just hoist the functionality out of them and into the base, then let it be configured with a bool. Sure you could create a *new* derived class that contains that functionality, but we’ve got an n-dimensional array of features here, one for each type, and there’s no way to turn it all into a nice tree. The model you choose to follow for your hierarchy design will end up being a prison. So to deal with this, you’ll probably end up turning that n-dimensional array into a list of configuration variables that sit in the base. And this isn’t necessarily a bad thing, though it usually is. Might as well have a single class with a pile of configuration variables and a bunch of switch statements, right?

Virtual override madness

The point of deriving is to specialize behavior. When you have derivatives of derivatives, you end up with potential ordering problems in your virtual function calls. A particular class overrides an OnDraw() method. It knows that it must draw its new stuff after the base class’s stuff, so it calls the base version first, then itself. A new derivative of this class wants to draw itself in between the base and its base. This is impossible, so you end up .

Increasing resistance to change

The more complex the system gets, the more paranoid people get about adding new things to it. If it’s a flat hierarchy, no big deal, just copy paste something else and adapt it. But it *will* have depth to it, and so when your junior guy wants to add some new gizmo to support a feature he’s working on, he freaks out! Where can it be inserted without messing everything up? Which virtual functions must you call and when and how? Most likely what they’ll do is add some more bools and put an if/else to stick in their new feature. Eventually you run into distributed state management problems, nasty.

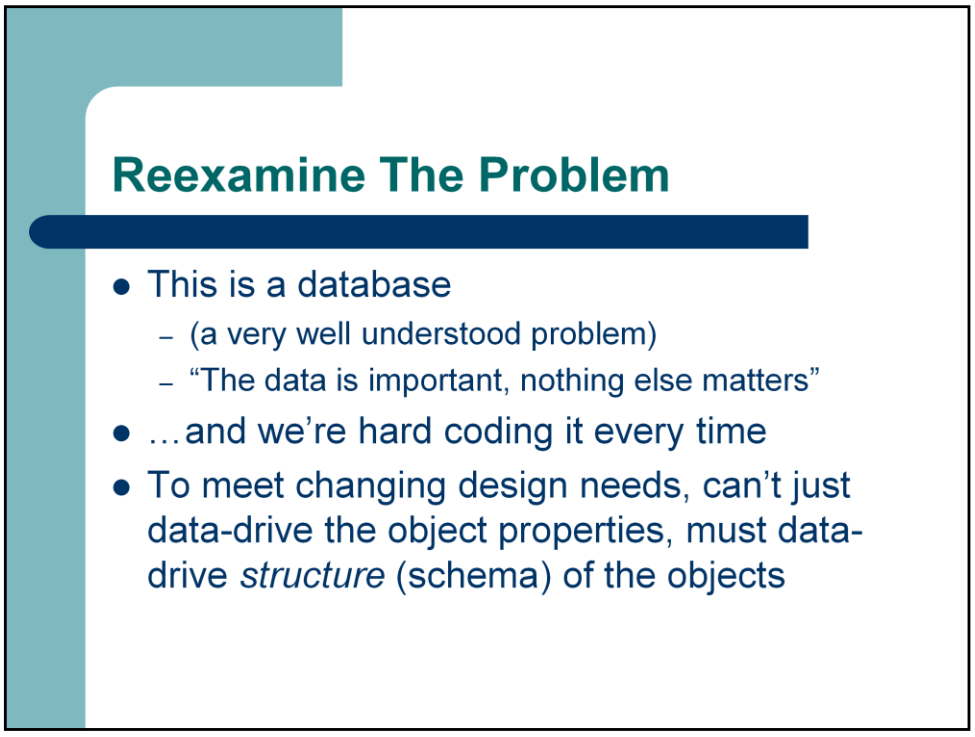
Doc rot

It’s hard enough to get programmers to document their own code properly, much less tell the rest of the team how to use it. In your data-driven system, somewhere there is some kind of loader function – code that maps data from the resource store onto runtime objects. The names of the fields it reads in, their

types, the allowable ranges, etc. are known as the “schema” of the game objects. Well since this schema is hard-coded into your C++ app, it’s up to the programmer to carefully document all of this separately, and keep it up to date as it changes. Just like the type problem, with code this close to the content, something this dynamic is going to take resources that probably aren’t there, and so documentation for the schema will end up limited or be subject to “doc rot”, and become more and more incorrect over time. This of course drives your scripters crazy.

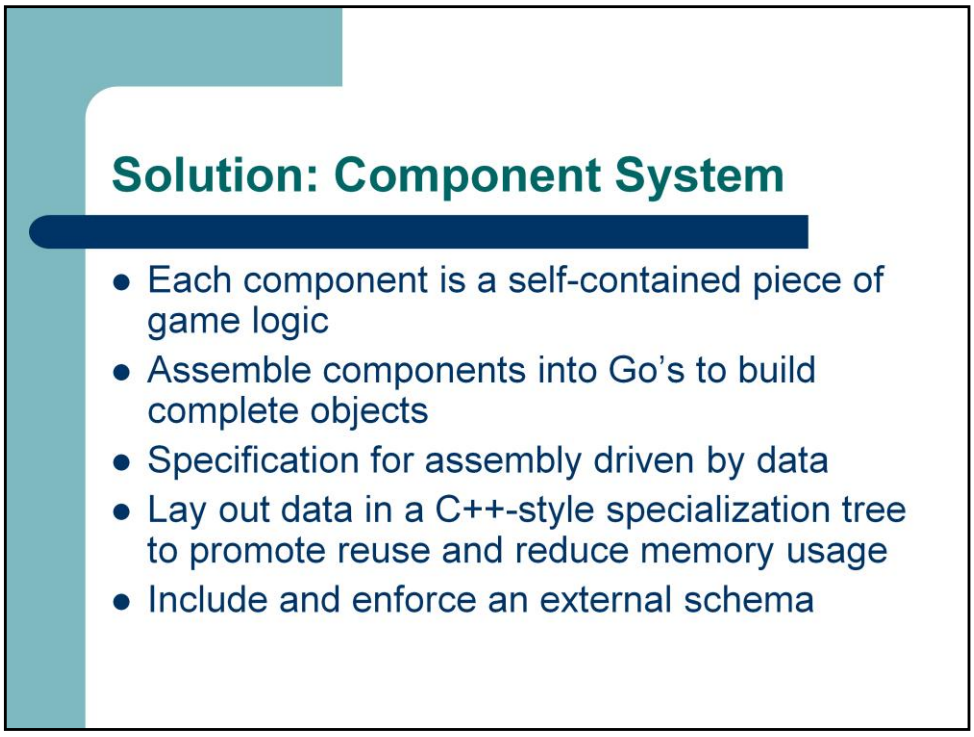
Editor out of sync

This is a similar problem to doc rot. Whatever tool you have built to place objects in a level needs to know the game schema in order to know what values it can let the designers change.



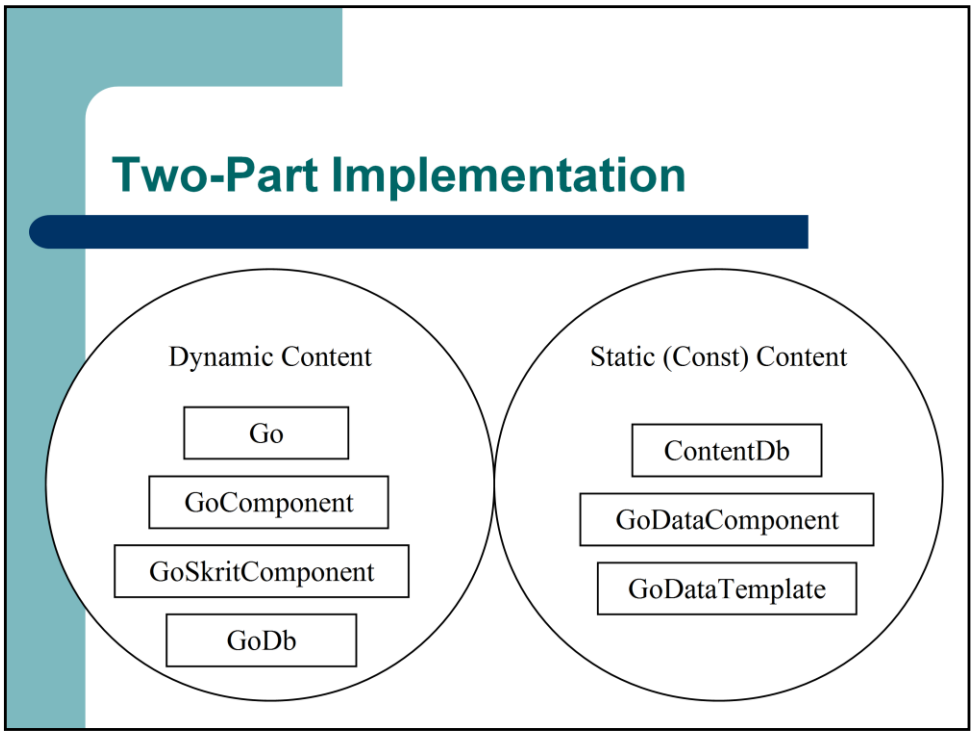
Let's step back a bit and look at what we're working on here. It's a database. Why are we spending all this time, constructing classes and hierarchies and managing cpp files and h files and #include nightmares and forward declarations and ordering dependencies on virtual functions and slow compile times and custom archive functions and all of this mess, when all we're really doing is hard-coding a database? Sure our game is data-driven in that we're reading in property values and such from disk in order to initialize our game objects, but in order to cope with the ever-changing needs of the game, the structure of the objects themselves is what must be data-driven now. Let's take the type structure, the hierarchy, and put it into data.

So if we're moving type definition out of the engine and into data, what should all of this look like? And now we're getting to the *real* point of this talk. The easiest way for me to describe this is to just talk about the system I implemented for Dungeon Siege – first implementation, then usage (next slide).

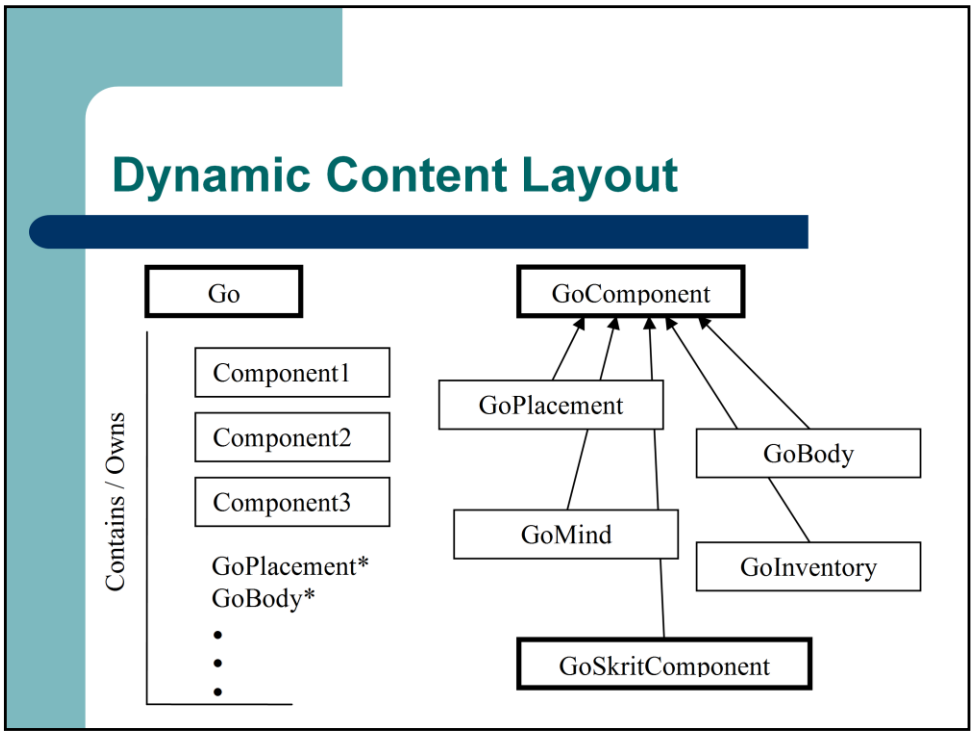


Here's a quick overview of the implementation. It's a simple component system, where each component encapsulates a chunk of game logic, and the data specifies how to assemble these components into Go's and what values they should be initialized with. If you don't like the word "component", try "plugin" instead, same thing.

For example, the [placement] component tracks the Go's location in the world, the [body] component is responsible for animation-related tasks, the [mind] component handles sensors and performing jobs, and the [inventory] component manages equipment and inventory items. These components are sort of like the mixin classes I mentioned earlier.



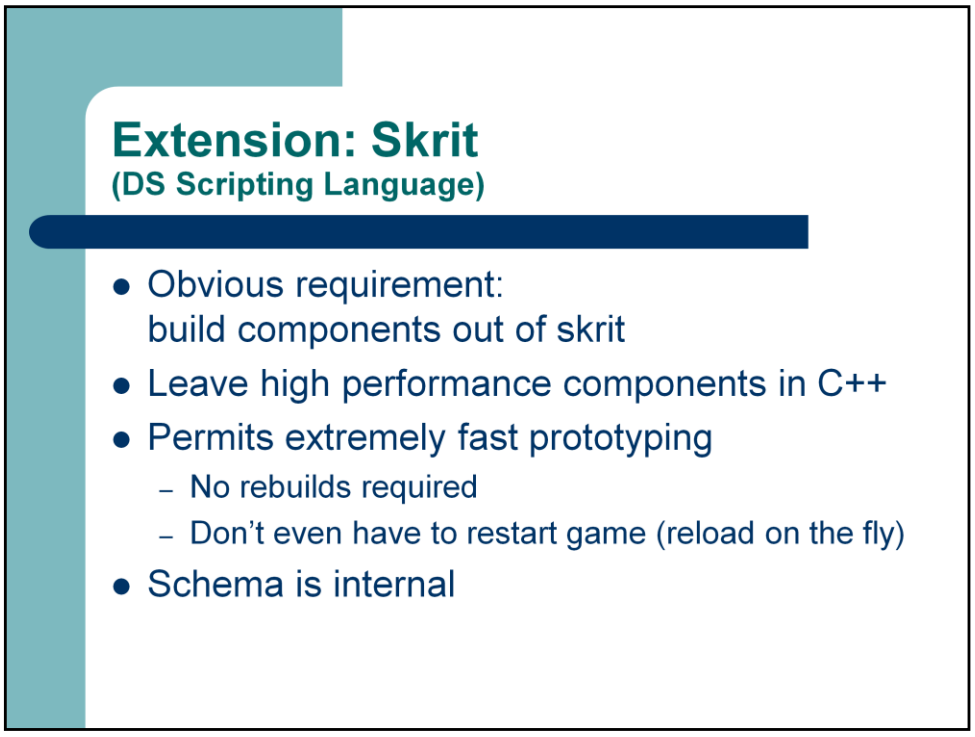
There are two separate sets of classes involved here. One is for the static content and represents the schema and prototypes in the database. The other is for dynamic content and makes up the game objects for the session. I'll be covering these two families of classes next.



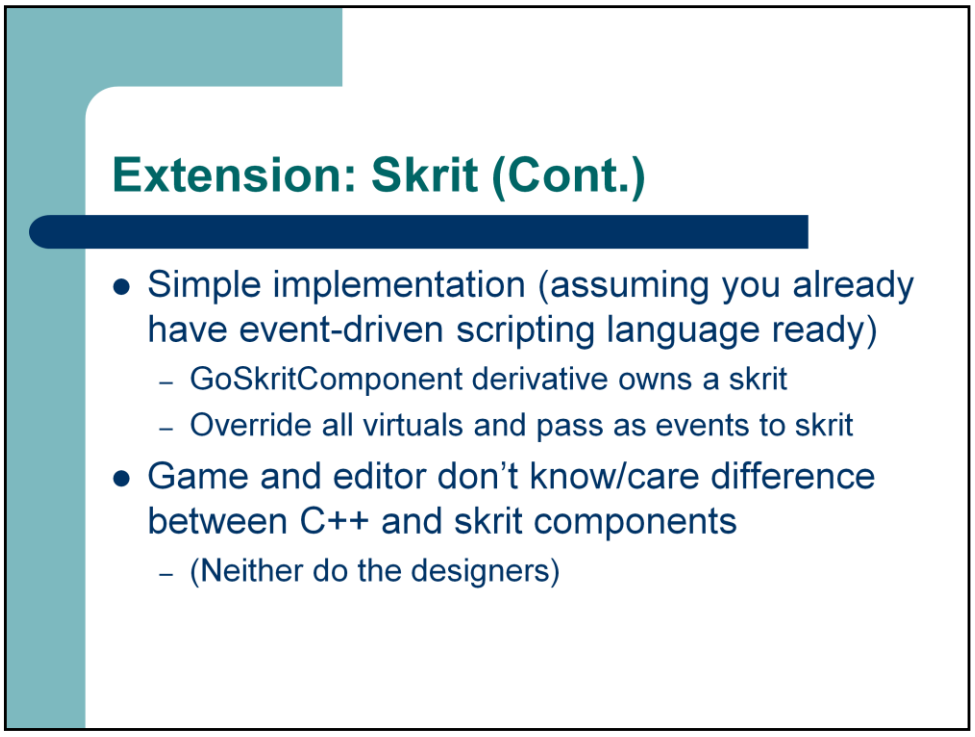
Let's look at the dynamic content first, which includes the Go's and components. I put thicker borders on the key classes in here. I also didn't show the GoDb because it's just a container for Go's.

A Go is just a class that contains a list of components. You can query for a component by name, and it will return a GoComponent* that you can cast to the real type. Components are unique within a Go, meaning that there can only be one of each kind. This is limitation I specifically added to keep the system simple but I plan to remove it for the next game as it turns out we needed to permit arrays and nesting in some cases. For performance, the most commonly used components get a cached pointer so we don't have to look them up each time (there are Get() functions for each of those, like GetBody(), GetInventory(), etc.). The Go class does not have derivatives. Nearly all logic is done in the components so the Go does little more than component management, maintain the parent/child tree, and a few other random things. The game runs almost completely on the logic built into the Go components.

GoComponent is our other main dynamic content class, and it's an abstract base class. It does little more than provide the common interface through which the system components communicate. It's filled with event handling methods meant to be overridden by derivatives, such as HandleMessage, CommitCreation, LinkParent, and Xfer (for persistence) and a few helpers. Most systems in the game that don't care about game code deal with components through virtual methods at the GoComponent level.

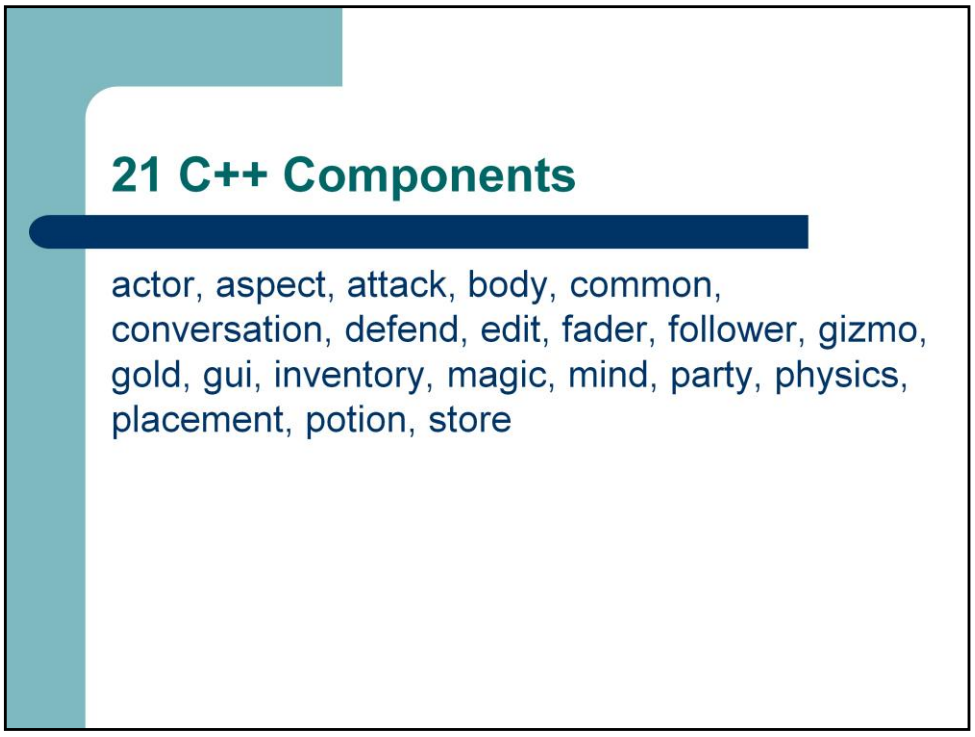


When I was designing this system it became immediately obvious that we wanted to build components out of Skrit (Skrit is just the name of my scripting language). High performance components that are used everywhere should be written in optimized C++ code, and everything else would be Skrit. Why do it like this? I wanted prototyping new ideas to be as simple as creating a new Skrit file and plugging it into the data somewhere. And because it's my compiler rather than Visual C++ I could do it all on the fly without having to restart the game (much less get a new build from an overworked engineer!).



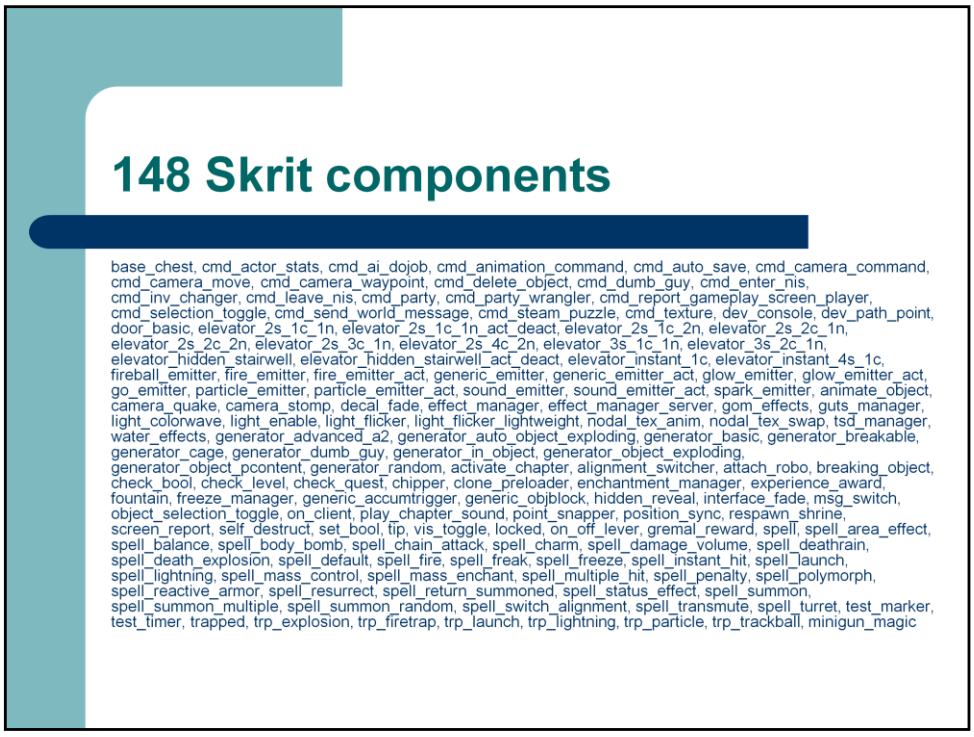
In practice this was pretty simple to implement. Just create a custom derivative of GoComponent called GoSkritComponent that owns a Skrit object, overrides all those virtual functions, and passes them along as events to the Skrit. For all practical purposes, the game and the editor don't know the difference between a C++ component and a Skrit component. And certainly none of the designers really had any idea either. They just see a property sheet in the editor.

It worked out pretty well. We ended up with about 21 components that we thought needed to be written in C++ (show next slide…)



(Mention a few of these).

And the rest, nearly 150 of them… (show next slide)

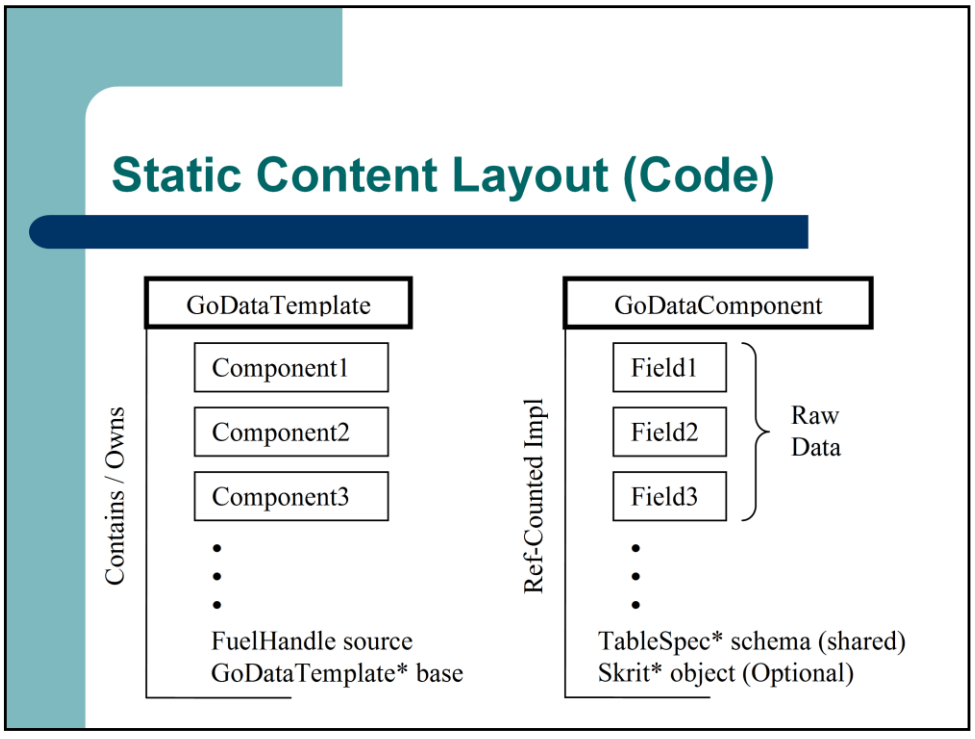


…ended up in Skrit. (“You probably can’t see these well, but…” and mention a few of them).

Looking back, we probably should have done even fewer in C++, but our content engineer didn’t come on board until we had already built a number of things in C++ so we just left most of them. We did end up converting a few C++ systems, such as our spellcasting, to use this Skrit component system though because it was so much easier.

Alert! Before Moving On

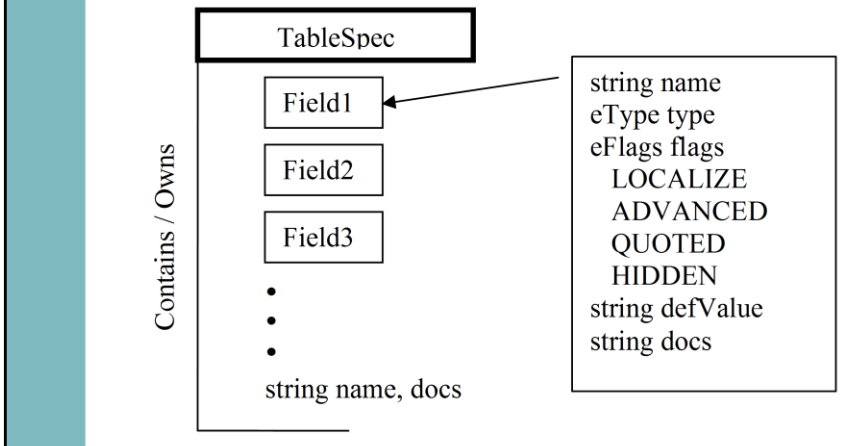
- Generic datastore required to continue
  - INI file, config file, XML store, RIFF, all the same
  - Permits generic data retrieval/storage
  - DS has “gas”, think “INI with nesting + goodies^2”
- Not difficult to roll your own
  - Many books/articles on this
  - Probably need one for other parts of the game anyway (i.e. you’ll find uses for it no problem)



(Template is a ‘pattern’ in the conventional meaning, not the C++ meaning.)

There is a 1:1 correspondence between a Go and its GoDataTemplate, and a Go’s components and the GoDataTemplate’s GoDataComponents. And within the GoDataComponents, the fields map 1:1 with the GoComponent public properties.

Schema Layout (Code)



Compile ContentDb
Part 1: Build Schema

1. Process components.gas (C++ table specs)
   a. Build table specs directly from .gas spec
2. Recursively scan components base directory for all skrit components
   a. Compile each skrit
   b. Build table specs from metadata in

…now we've got the schema constructed.

C++ Component Schema (Data)

```
[t:component,n:gui]
{
    doc = "Objects with GUI may be placed in inventory";
    required_component* = aspect;

    [inventory_icon]
    {
        type = string;   default = b_gui_ig_i_it_def;
        doc = "Bitmap displayed when dragging or in inventory";
    }
    [active_icon]
    {
        type = string;   default = b_gui_ig_i_ic_def;
        doc = "Bitmap displayed in quick-select window";
    }
    ...
}
```

Talk about how it's important to have a public schema like this.

Skrit Component Schema (Data)

(Concept adapted from UnrealScript)

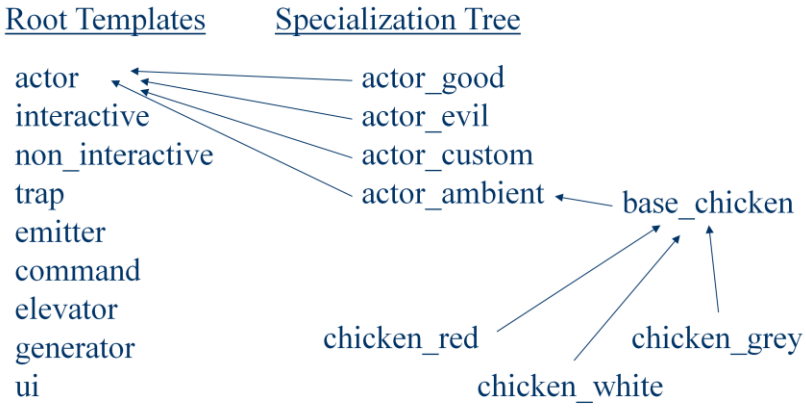
```
property string effect_script$      = ""
doc = "Name of the SiegeFx script that will be providing the visual.";
property string end_script$         = ""
doc = "Name of the SiegeFx script that will be providing the visual "
      "when un_summoning.";
property string script_params$      = ""
doc = "Parameters to send to SiegeFx script";
property string template_name$      = ""
doc = "Template name of actor to summon";
property string state_name$         = "summoned"
doc = "Name of effect to use as a generic state and as a screen name.";
property string description$        = ""
doc = "Description of enchantment being applied";
property string caster_description$ = ""
doc = "Description of enchantment being applied to the caster";
property bool   guard_caster$       = true
doc = "Make the summoned creature follow the caster.";
property bool   change_align$       = true
doc = "set summon alignment to be that of the caster.";
property bool   delete_inv$         = true
doc = "delete summons inventory when removed.";
```

Compile ContentDb
Part 2: Build Templates

(This is just prep work)

1. Recursively scan .gas template tree
   a. Note: doesn't need to be a physical tree
2. Open data handles to each template
3. Keep track of root nodes, build specialization tree

Template Forest (Data)



Template Specification (Data)

```
[t:template,n:chicken_white]
{
    category_name = "1w_ambients";
    doc = "chicken_white";
    specializes = base_chicken;
    [aspect]
    {
        [textures] { 0=b_c_na_ckn_white; }
    }
    [common]
    {
        [template_triggers]
        {
            [*]
            {
                action* = call_sfx_script("feathers_flap_white");
                condition* = receive_world_message("we_anim_sfx",1);
            }
        }
    }
    [physics]
    {
        break_effect = feathers_white;
        explode_when_killed = true;
    }
}
```

Implicit in this template is the other components that are inherited from the base template. There are many that you don't see here like physics (yes, chickens can simulate) and actor.

Compile ContentDb

Part 3: Compile Templates

1. Recursively compile templates root-down
2. Add data components on demand
3. Read in values, override base template fields

This is all similar to C++ base-first member initialization in ctors.

Compile ContentDb
Special notes

- We want a flat tree for performance reasons
  - Depends on how frequently you construct objects and how fast your data override system is
  - Also permits special const-read optimization that can eliminate memory usage and CPU for variables that are never changed
- Copy data components on write to avoid unnecessary memory usage
- If have many templates, will need to JIT compile leaf templates to save memory

Editor Integration

- This is *almost* trivial
- Editor should have a property sheet type thing
  - This is a one-entry view into the db
  - Map types and names onto fields using schema
  - Can un-override easily by querying template
  - Be sure to add a column or tooltip for docs!

Editor Integration (Cont.)

- For DS all editing support done through a special “GoEdit” component
  - Transforms data between game object and editor
  - Supports cheap rollback (undo) by double buffering
  - Does not exist in game, only needed in editor
  - Automates saving all game object instances – just compare vs. the const data and write out if different
- Not recommended: permitting forced overrides of duplicate data

Instance Specification (Data)

```
[t:chicken_red,n:0x837FD928]
{
    [placement]
    {
      p position = 1.3,0,1.8,0x1738FFDB;
      q orientation = 0.3828,0.2384,-0.7772,0.98;
    }
    [common]
    {
        screen_name = "Super Chicken";
    }
    [body]
    {
        avg_move_velocity = 18.000000;
    }
}
```

Loading Objects

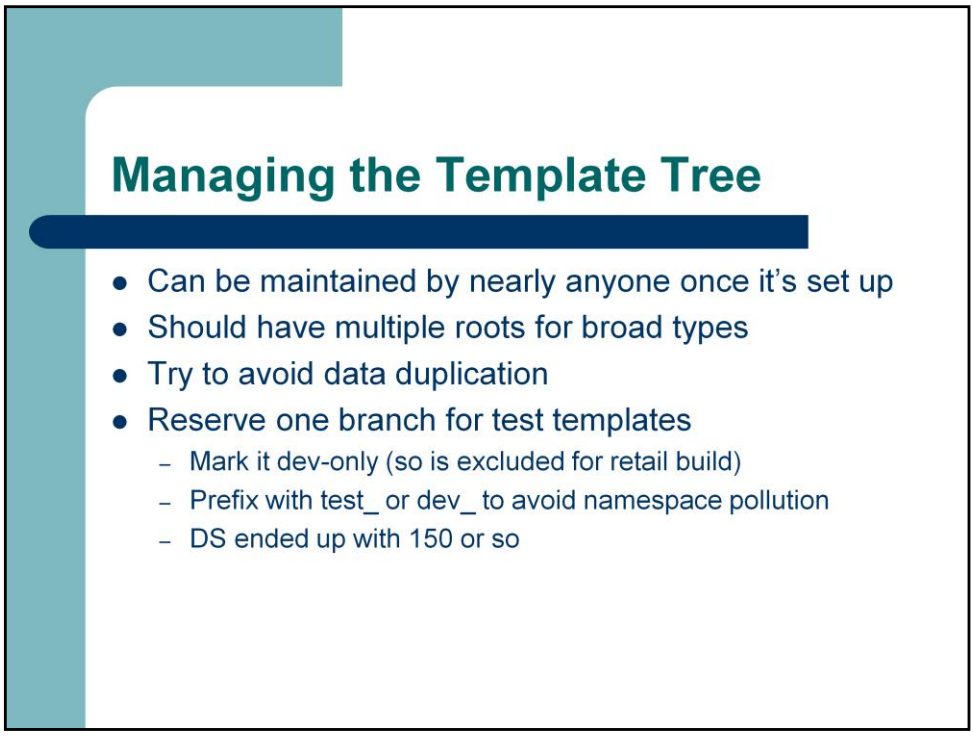
- In DS, objects are referenced by content ID
- Look up instance block to get template to use
- Instantiate Go by that template
  - For each block in instance, create a new data component
  - Specialize that data component from base in template
  - Finally iterate through GoComponents and xfer in data to set initial values

New C++ Components

- Can be done with little regard for other components (just add it)
- Derive from GoComponent *only*
  - Specializing an existing class just asking for trouble
- Add new block to C++ components schema (DOC IT)
- Use a factory method
  - Simple LUT mapping name ➲ 'new GoJooky'
- Wait a second, wouldn't it be better to write using the scripting language? (Probably…)

New Skrit Components

- Same as C++, just stick it in there
- Everything should be autodetect here
- Extend the scripting language with metadata
  - Pass it straight through to schema query
  - Can implement flags, docs, and custom game features like “server only” components etc.



Recommendation: have a special part of the tree (make it compile out in retail builds) that is just for test objects. They'll pollute the global namespace so prefix the names with something like test_ or dev_. Then you can let people work in that branch of the tree doing whatever they like for testing purposes without worrying about it screwing up the main game. DS ended up with nearly 150 of these 'cause whatever, they're just for testing…

Advantages I Forgot To Mention

- Direct and automatic editor support
- Designers can construct their own types to place in the editor (careful, monitor this!)
- By only saving out modified data in instances, can make global changes easily by modifying templates
- Reorganizing the template tree is easy
- If embed a sub-tree for designers to build custom views into the database

Some Pitfalls

- C++ components prone to becoming intertwined
  - Operations can end up being order-dependent, though this is more easily controlled
  - Nothing here is unique to components
- It's a little *too* easy to add templates, perhaps
  - DS has >7300 of them, many auto-generated
  - System was designed for <100
  - Need to keep close eye on template complexity to avoid memory/CPU hog (i.e. unnecessary components or wacky specialization)
- "With power comes responsibility"

Future

- Schema extensible
- Add flags and constraints that editor can use
  - Auto-detect when can use color chooser or slider or listbox or whatever
- Add defaults computed from script

A paper is not available, unfortunately shipping Dungeon Siege took up all my time. ☹