A Data-Driven Game Object System

GDC 2002

Scott Bilas
Gas Powered Games

Introduction

- Me
  - Scott Bilas
  - Background
- You
  - System architect types
  - Tired of fighting with statically typed systems for game code
- The Test Subject
  - Dungeon Siege
  - >7300 unique object types (i.e. can be placed in the editor)
  - >100000 objects placed in our two maps
  - Continuous world means anything can load at any time

Cell Phones?

Definitions

- Data-Driven
  - Meaning: “No engineer required”
  - Engineers are slow
  - Causes designers to hack around missing functionality
  - Goal: remove C/C++ from game
  - Line between engine and content is always moving

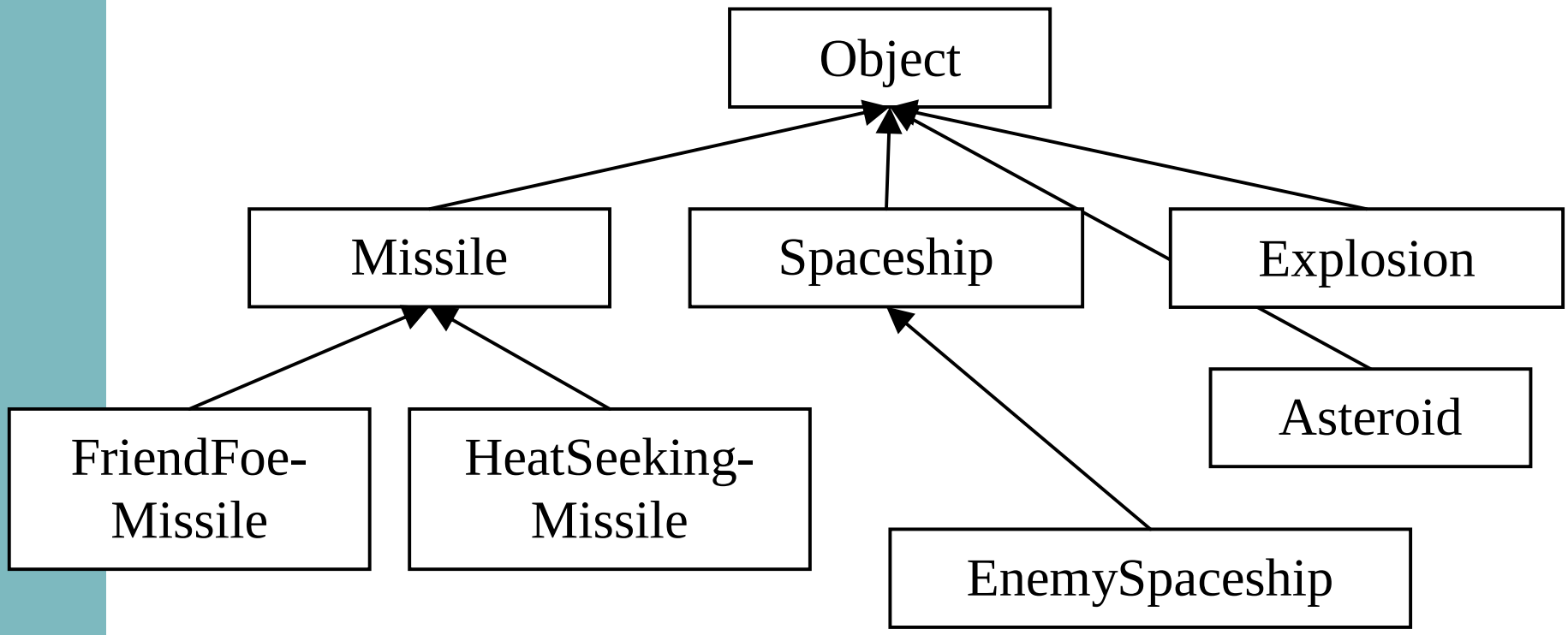
Definitions (Cont.)

- Game Object (Go)
  - Piece of logical interactive content
  - Perform tasks like rendering, path finding, path following, speaking, animating, persisting
  - Examples are trees, bushes, monsters, levers, waypoint markers, doors, heroes, inventory items
  - Many are “pure logic”, never see them (triggers, elevator movers, camera sequences)
  - Every game has these in some form

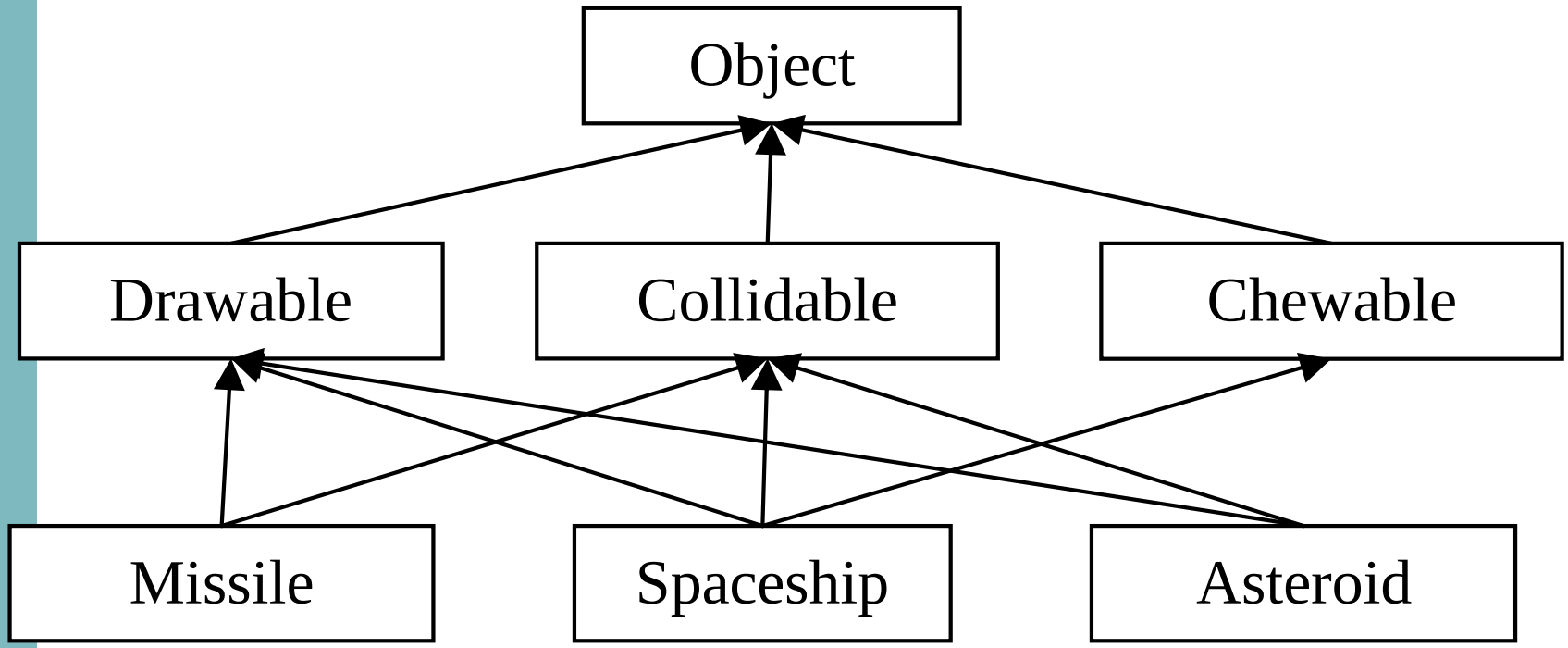
Definitions (Cont.)

- Game Object System
  - Constructs and manages Go's
  - Maps ID's to object pointers
  - Routes messages
  - Build from many things, but for this talk
    - GoDb: Go database
    - ContentDb: Static content database
  - Every game has this in some form

Example Class Tree
Vintage



Example Class Tree
Newfangled



It Won't Work

- There are hundreds of ways to decompose the Go system problem into classes
  - They are all wrong
  - They don't start out wrong, of course…
- Games constantly change
  - Designer makes decisions independently of engineering type structures
  - They *will* ask for things that cut right across engineering concerns

Just Give In To Change

- Requirements get fuzzier the closer your code gets to the content
- Will end up regularly refactoring
- Do not resist, will cause worse problems!
- However: C++ does not support this very well!!

C++: Not Flexible Enough

- Code has a tendency to “harden”
  - Resists change over time
  - Rearranging class tree requires lots of work
- Needing to change it causes engineering frustration, which leads to…
  - Class merging/hoisting (fights clean OOP)
  - Virtual override madness
  - Increased complexity ➲ increasing resistance
  - Doc rot, editor out of sync

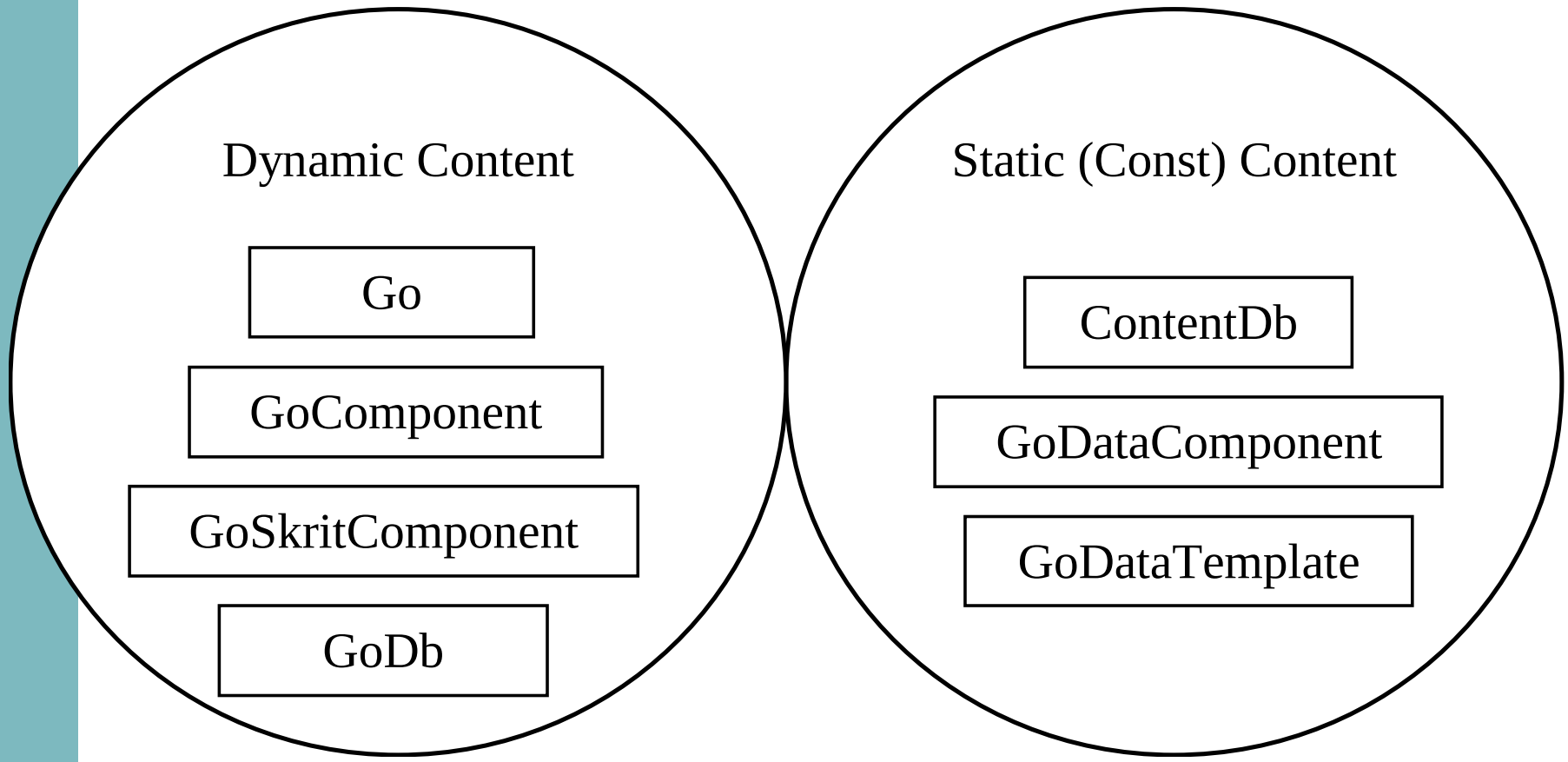
Reexamine The Problem

- This is a database
  - (a very well understood problem)
  - “The data is important, nothing else matters”
- …and we’re hard coding it every time
- To meet changing design needs, can’t just data-drive the object properties, must data-drive *structure* (schema) of the objects

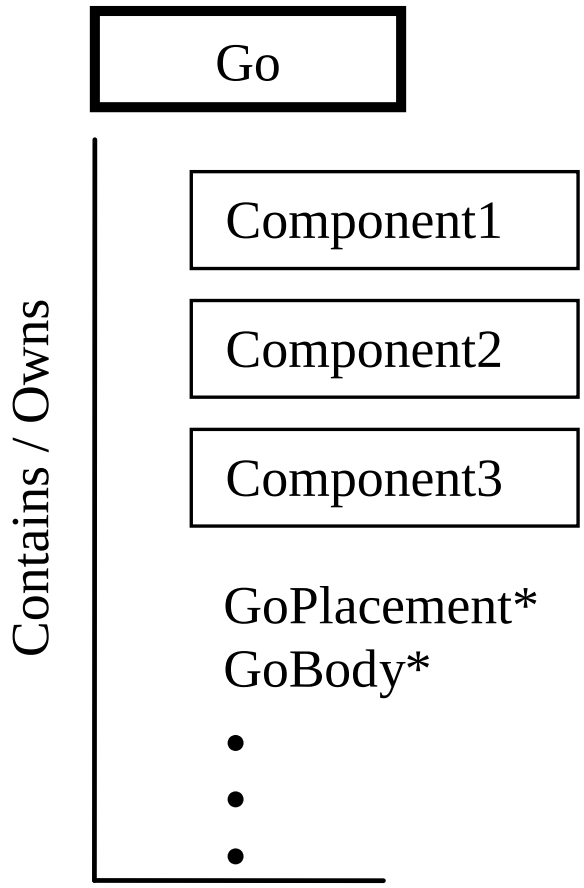
Solution: Component System

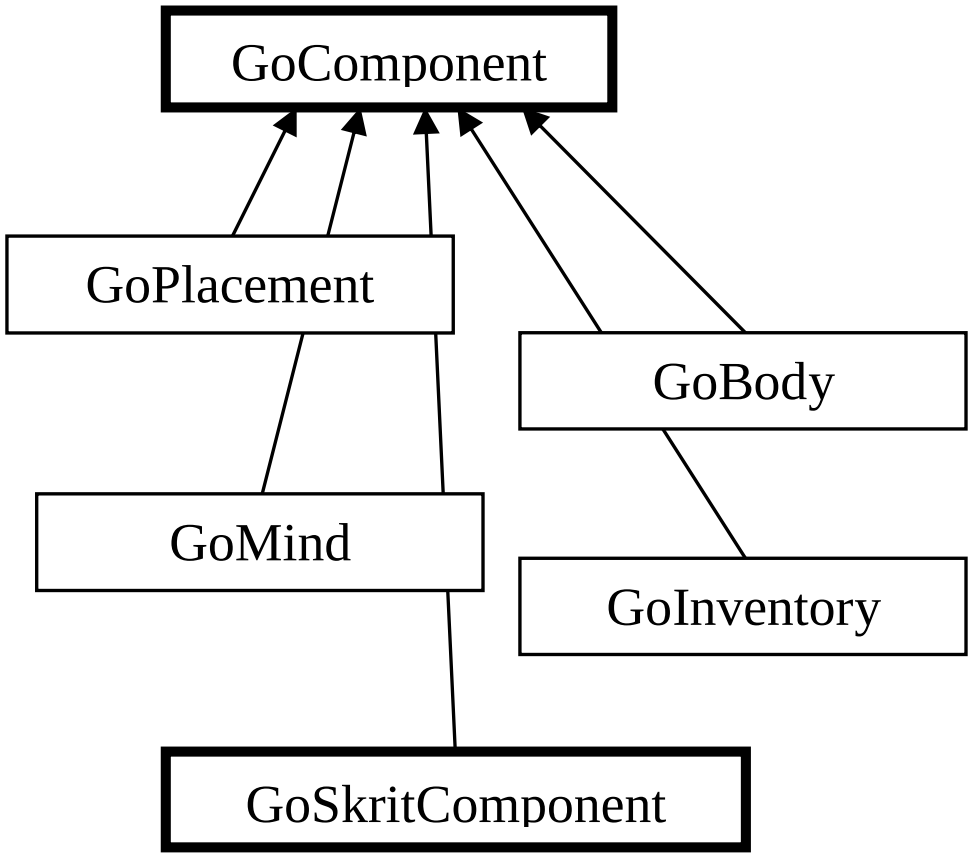
- Each component is a self-contained piece of game logic
- Assemble components into Go's to build complete objects
- Specification for assembly driven by data
- Lay out data in a C++-style specialization tree to promote reuse and reduce memory usage
- Include and enforce an external schema

Two-Part Implementation



Dynamic Content Layout





Extension: Skrit
(DS Scripting Language)

- Obvious requirement:
  build components out of skrit
- Leave high performance components in C++
- Permits extremely fast prototyping
  - No rebuilds required
  - Don’t even have to restart game (reload on the fly)
- Schema is internal

Extension: Skrit (Cont.)

- Simple implementation (assuming you already have event-driven scripting language ready)
  - GoSkritComponent derivative owns a skrit
  - Override all virtuals and pass as events to skrit
- Game and editor don’t know/care difference between C++ and skrit components
  - (Neither do the designers)

21 C++ Components

actor, aspect, attack, body, common, conversation, defend, edit, fader, follower, gizmo, gold, gui, inventory, magic, mind, party, physics, placement, potion, store

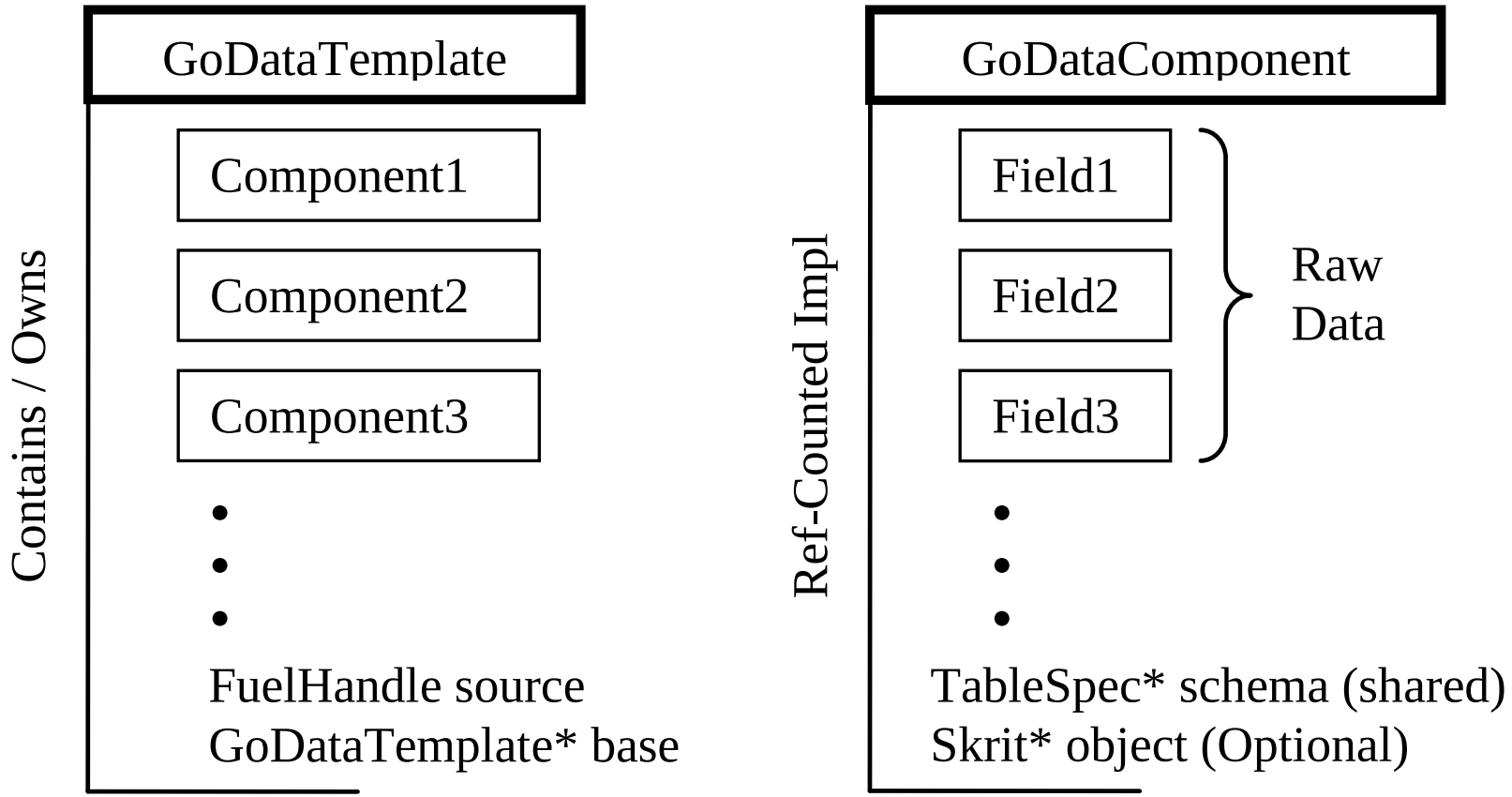
148 Skrit components

base_chest, cmd_actor_stats, cmd_ai_dojob, cmd_animation_command, cmd_auto_save, cmd_camera_command, cmd_camera_move, cmd_camera_waypoint, cmd_delete_object, cmd_dumb_guy, cmd_enter_nis, cmd_inv_changer, cmd_leave_nis, cmd_party, cmd_party_wrangler, cmd_report_gameplay_screen_player, cmd_selection_toggle, cmd_send_world_message, cmd_steam_puzzle, cmd_texture, dev_console, dev_path_point, door_basic, elevator_2s_1c_1n, elevator_2s_1c_1n_act_deact, elevator_2s_1c_2n, elevator_2s_2c_1n, elevator_2s_2c_2n, elevator_2s_3c_1n, elevator_2s_4c_2n, elevator_3s_1c_1n, elevator_3s_2c_1n, elevator_hidden_stairwell, elevator_hidden_stairwell_act_deact, elevator_instant_1c, elevator_instant_4s_1c, fireball_emitter, fire_emitter, fire_emitter_act, generic_emitter, generic_emitter_act, glow_emitter, glow_emitter_act, go_emitter, particle_emitter, particle_emitter_act, sound_emitter, sound_emitter_act, spark_emitter, animate_object, camera_quake, camera_stomp, decal_fade, effect_manager, effect_manager_server, gom_effects, guts_manager, light_colorwave, light_enable, light_flicker, light_flicker_lightweight, nodal_tex_anim, nodal_tex_swap, tsd_manager, water_effects, generator_advanced_a2, generator_auto_object_exploding, generator_basic, generator_breakable, generator_cage, generator_dumb_guy, generator_in_object, generator_object_exploding, generator_object_pcontent, generator_random, activate_chapter, alignment_switcher, attach_robo, breaking_object, check_bool, check_level, check_quest, chipper, clone_preloader, enchantment_manager, experience_award, fountain, freeze_manager, generic_accumtrigger, generic_objblock, hidden_reveal, interface_fade, msg_switch, object_selection_toggle, on_client, play_chapter_sound, point_snapper, position_sync, respawn_shrine, screen_report, self_destruct, set_bool, tip, vis_toggle, locked, on_off_lever, gremal_reward, spell, spell_area_effect, spell_balance, spell_body_bomb, spell_chain_attack, spell_charm, spell_damage_volume, spell_deathrain, spell_death_explosion, spell_default, spell_fire, spell_freak, spell_freeze, spell_instant_hit, spell_launch, spell_lightning, spell_mass_control, spell_mass_enchant, spell_multiple_hit, spell_penalty, spell_polymorph, spell_reactive_armor, spell_resurrect, spell_return_summoned, spell_status_effect, spell_summon, spell_summon_multiple, spell_summon_random, spell_switch_alignment, spell_transmute, spell_turret, test_marker, test_timer, trapped, trp_explosion, trp_firetrap, trp_launch, trp_lightning, trp_particle, trp_trackball, minigun_magic

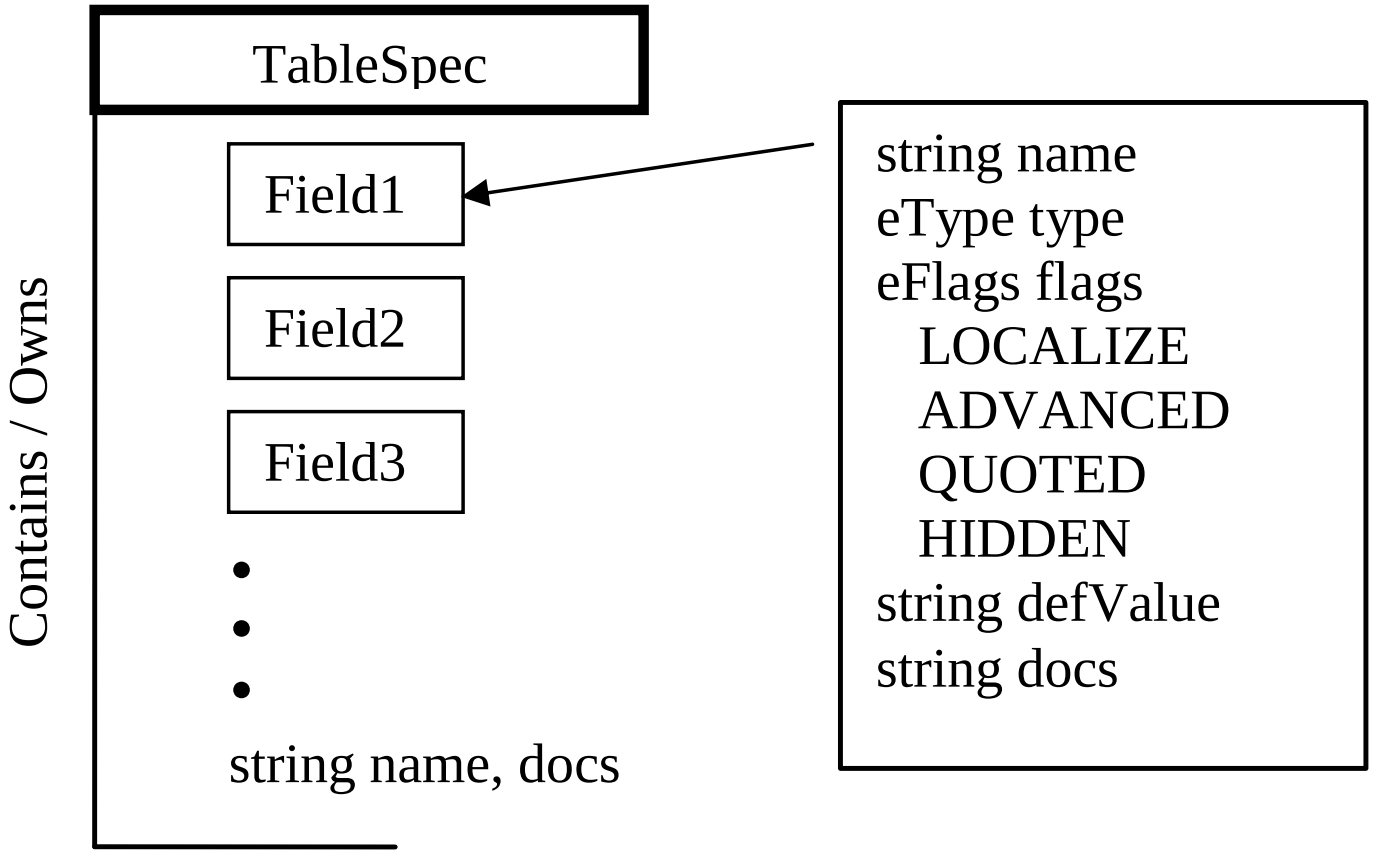
Alert! Before Moving On

- 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)

Static Content Layout (Code)



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";
    }
    ...
}
```

Skrit Component Schema (Data)
(Concept adapted from UnrealScript)

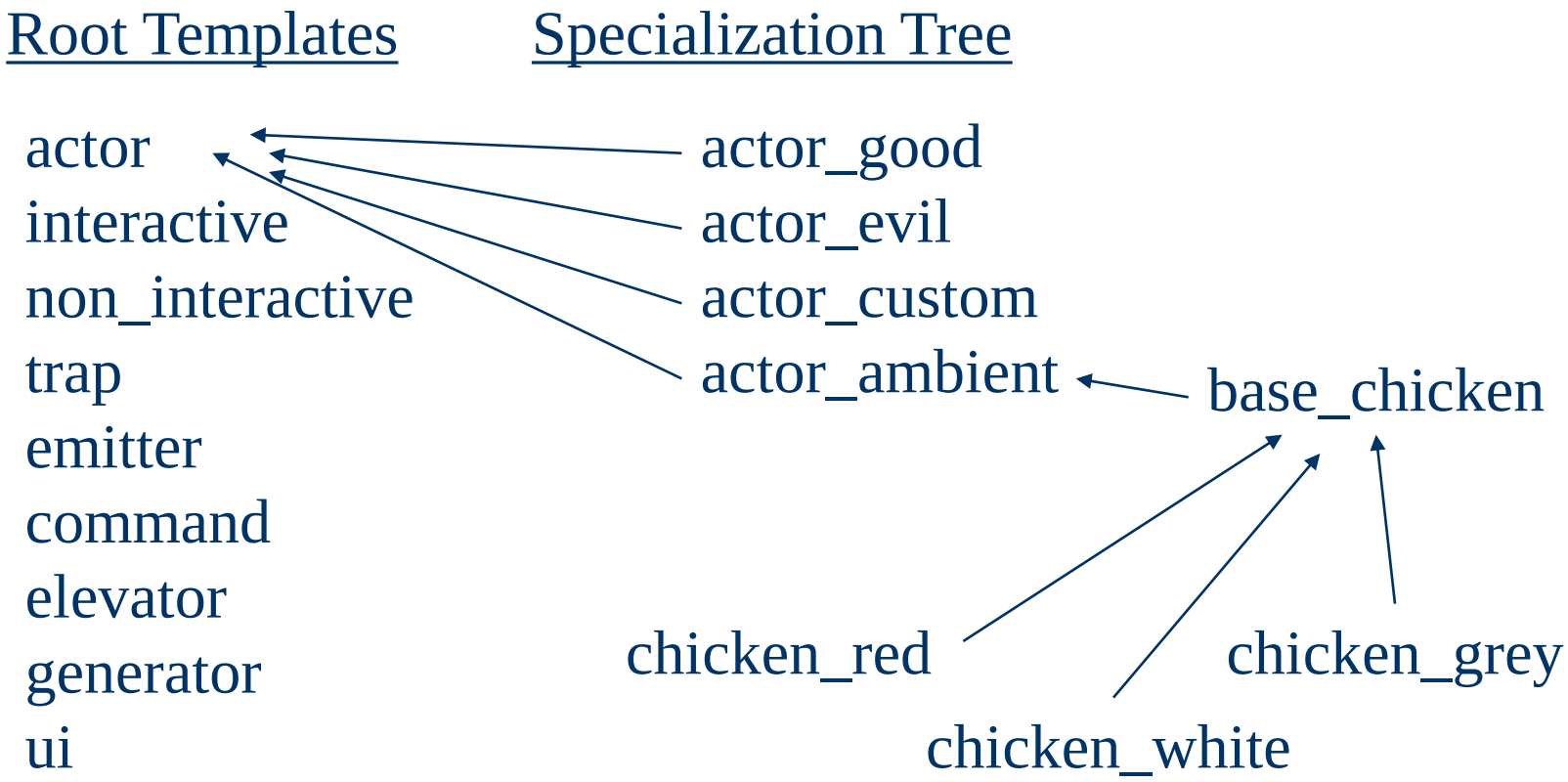
```
property string effect_script$      = ""
doc = "Name of the SiegeFx script that will be providing the visual.";
property string end_script$         = ""
doc = "Name of the SiegeFx script that will be providing the visual “
      "when un_summoning.";
property string script_params$      = ""
doc = "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;
    }
}
```

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.

Managing the Template Tree

- Can be maintained by nearly anyone once it's set up
- Should have multiple roots for broad types
- Try to avoid data duplication
- Reserve one branch for test templates
  - Mark it dev-only (so is excluded for retail build)
  - Prefix with test_ or dev_ to avoid namespace pollution
  - DS ended up with 150 or so

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

Contact Info

Scott Bilas

http://scottbilas.com