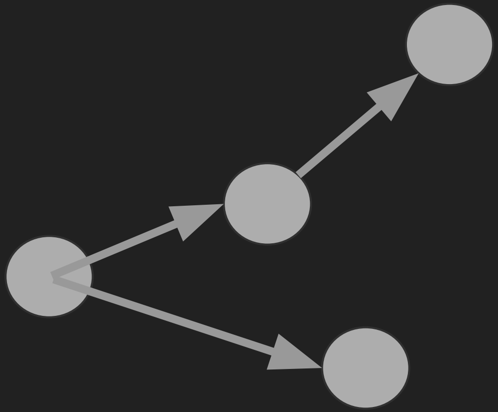
Building Systems

Object Graphs

- Objects
- Reference Types and Value Types
- A Tree of Objects



A Model for your Program

- This can be mutable or immutable
- It serves as a handle to the state of your program
- This handle can be fed to different parts of your program
- A “Controller” patch now can change the state
- A “View” patch can visualize the state of your program

RECORD

An Immutable Model

- Let’s recall the advantages of immutability

  SnapShots

  - Snapshots can recorded
  - You easily can Undo changes
  - You could jump to different Snapshots or lerp between them
  - You can detect Changes
  - Cache regions are compatible with this Snapshot idea

RECORD

Let’s have a look at an Example

- A controller wants to CHANGE the state of a program
- How do we do that?
- We need a way of changing the CURRENT Snapshot
- So we need some idea of a changing system
- Woops: that sounds like mutation
- But didn’t we want immutability?
- We want both!
- Let’s have a look at a mutating Program based on Snapshots

RECORD

A Mutable Model

- My state changes each frame
- I still want to distribute that state to my Views
- My controlling patches are Inputs to the System
  - Designer-Patches
  - Patches that receive recieve data from the network
- I want to “normalize”/”fix” my data before it gets rooted to my View patches
- And it feels wrong to think in Snapshots.
- For me it’s just a program state that mutates along

CLASS

We could even combine these ideas

- A mutating AppState: a Class
- We don’t make use of Reference<T>, as we have our own mutating AppState Class

- It has all sorts of mutating data, structured in several classes
- Somewhere in there can be immutable Snapshots

Composition pattern

- We have a object graph again
- This time less typed
- This way we can have a “user” of our system composing the graph in a way we didn’t foresee
- We need a glue between the objects in the graph
- A contract
- An abstract data-type
- An interface

Composition pattern

- We implement the interface on different classes
- Again we make use of the Process node feature to make it easy to create instances of those classes
- All of them have an Output that is compatible to the glue, the interface

Entity/Component pattern

- Entities now form our object graph
- But Entites can be stupid
- They are only objects that can be enriched via Components
- This results in a even greater flexibility
- Entities CAN be intelligent
- The main point is that users can attach information to them
- Your system now is even more customizable by the user

Strategy pattern

- Think of an automata
- Only one State/Mode is active
- We want to transition to another Mode anytime
- We want that little machine to know the current Mode
- We want to get rid of the old Mode
- Only the current Mode shall be updated and cost resources
- By this we have very lightweight machine