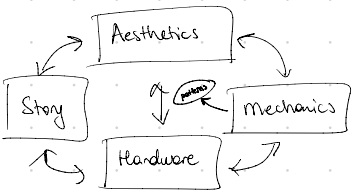
game programming patterns



- at least two different patterns to test at the end.

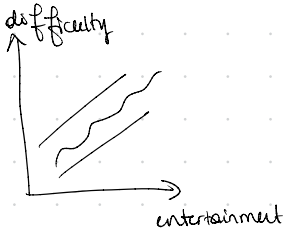
* get information
* find a game idea
* brainstorming
  - → mechanics
  - → story
  - → characters
  - → genre
  - → environment

* have too many ideas at the beginning
* get inspiration from different media
* be harsh in your evaluation
* storyboards, sketches
* don't be judgemental
* know your resources and skillset
* don't duplicate
* enjoy the process
* be creative
* patterns – have a rough look & mention at least two of them.
* don't rush
* literature survey.

- genre: X
- word: ...
- characters: Y and Z
- platform: ...
- story: ...
- technology: ...

what is the player doing?
→ player-centric approach

* focus on protagonist.



14.03.2024

Introduction

beginning of programming patterns

* Tunnels & Trolls
* BASIC
* Design Patterns: Elements of Reusable... (?)
* Gang of Four.

! Singleton pattern

No lecture on 21.03.2024

28.03.2024

* instead of writing code → organising the code
* design patterns come into play
* Domain specific books, whole engine books are not well enough to capture general idea.

→ **Design Pattern for Programming**

we are looking this in a gaming point of view.

1. time and Sequencing
2. interactions
3. performance

i.e creating forests,(1) actors to observe the environment,(2) refreshing the environment without crashing(3)

- good architecture

? what is a good architecture ?

- decoupling - seperate modules

? what is the cost of decoupling? — should always consider this.

- the bad in the good intentions : sometimes bad codes are better.

Programming as a Flowchart

- abstractions and decoupling.

- Performance and ?s
- Iterations & expectation
- prototyping

Balance in Games

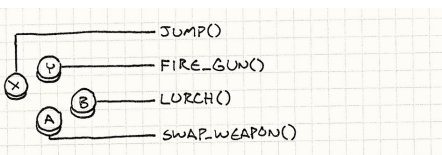
X no dominant strategy

? what do we want ?

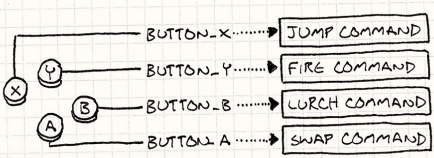
- nice architecture
- fast run-time performance
- get today's feature ? ? ?
- simplicity

* Command Pattern

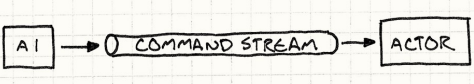
user input, buttons and actions



↳
- button pressed
- keyboard events
- mouse clicks



↳
- independent of the actor
- personalised input setting can be possible.

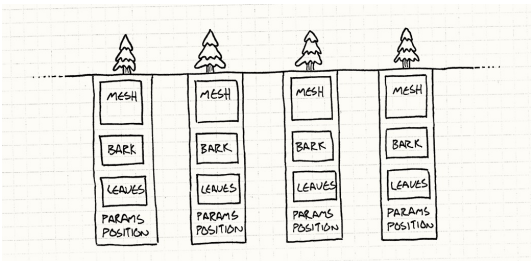


↳ we can undo and redo by this.

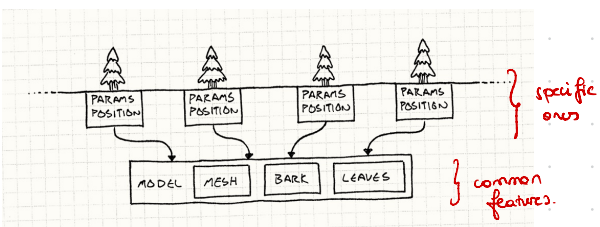
* Flyweight Pattern (lightweight)

Thousands of trees

Each tree has a mesh of polygons
texture
location and orientation
tuning parameters



↳ repeating everything...

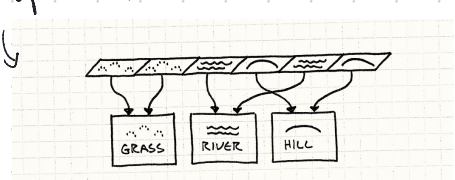


↳ cloning common items and create a model

? GPU instancing ?

* For generic items, we can create different types of instances.

↳

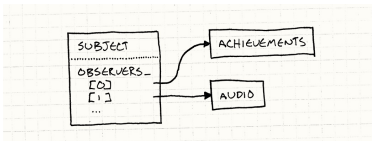


* Observer Pattern

achievements

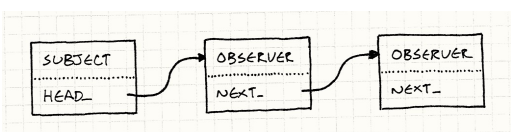
↳ grabbing an object
↳ killing an enemy
↳ etc.

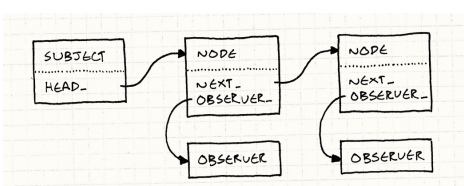
- good & very powerful abstraction.
- one piece of code to announce that smth. interesting has happened



some people say
- too slow
- too fast
- dynamic allocation?

* we can have linked observers





↳ a pool of list nodes

add a Garbage Collector

↳ if there is a bug, it is a huge problem

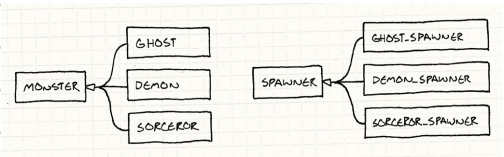
this is important

⇒ get notified when state has changed.

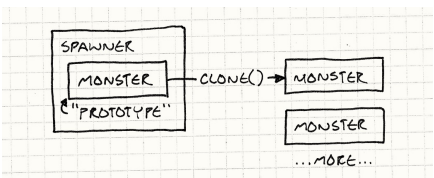
! state pattern comes into play

* Prototype Pattern

spawner for each type of monsters

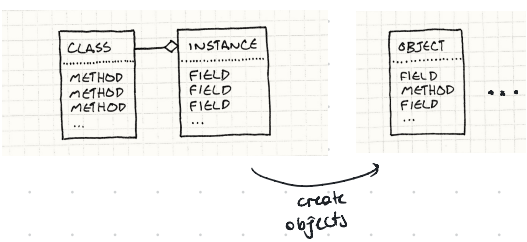


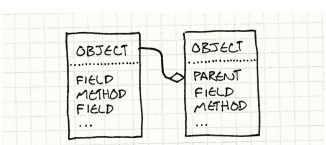
* we can create a spawner prototype



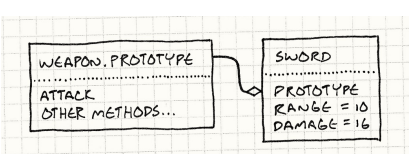
and generate several different monsters.

→ Prototype language Paradigm





* Javascript ?



* Singleton Pattern

- it started with a naive idea but people use it too much and it became a garbage
- one object, one instance ⇒ reaches from GLOBAL ⇐ every where

* manager class → you can access easily

! why we use it ?

→ it doesn't create the instance if no one uses it
→ it is initialised at run-time
→ you can sub-class it.

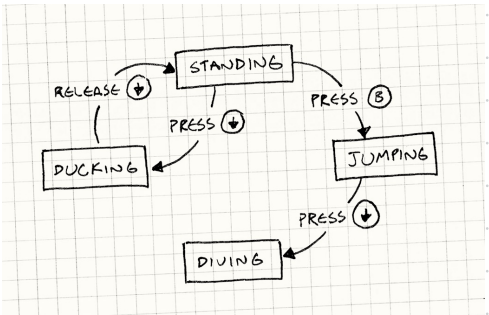
! why regret using it ?

→ it is a global variable.
→ they make it harder to reason about the code
→ it encourages coupling
→ they are not concurrency-friendly
→ they solve two problems when you have only one
→ lazy initialisation takes control away from you

! what we can do instead ?

→ no manager, no problem
→ limit a class to a single instance
→ provide convenient access to an instance
→ pass the object you need as an object
→ get it from the base class
→ get it from smth already global
→ get it from a service locator

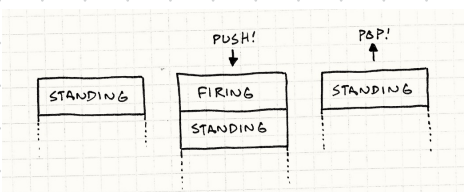
* State Pattern



→ we can use enums & switches

→ superstates & substates

* push-down automata



Sequencing Patterns

26.04.2024

Design patterns related on time

* Double buffer
* Game loop
* Update method

* Double Buffer

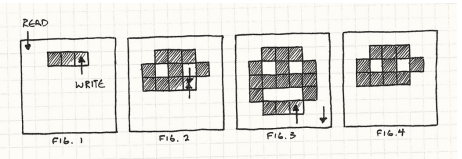
Framebuffer (in code)

Two buffers → swap this.

* How we can make things looks smoothly in a fast way?

- instantaneous
- simultaneous

Rendering



writing and reading at the same time. How to optimise this?

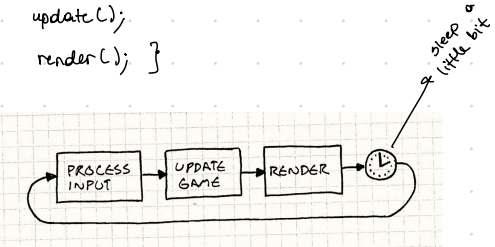
We're going to use 2 Framebuffers.

- When to use it?
- How?

using pointers (faster, less memory)

* Game Loop

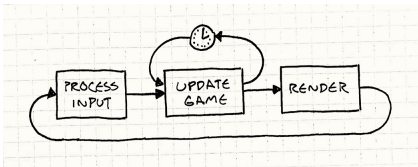
```
while (true) {
    processInput();
    update();
    render(); }
```

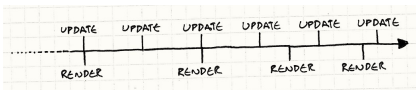


why don't we add something like this:

```
double lastTime = getCurrentTime();
while (true) {
    double current = getCurrentTime();
    double elapse = current - lastTime;
    :
    lastTime = current }
```

02.05.2024





will going to be the best solution.

Behavioural Patterns

- Bytecode
- Sandbox
- Type Object

1 Bytecode

Give behavior the flexibility of data by encoding it as instructions for a virtual machine.

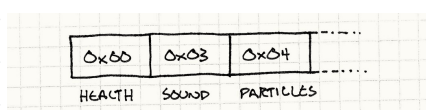
like a tree, lots of arrows, hard to update

traversing is hard and it is slow.

→ writing as a virtual machine, instead of running the actual code, create a virtual machine.

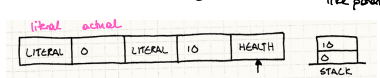
→ create a low level instruction stack and run it on VM

* when we have lots of behaviours and game language does not make it possible to run them, we create our scripting system, like all things.
  → we'll need front-end
  → miss the debugger.



initialisations will be low level, but afterwards it become high level

→ a stack machine including instructions top of each other to how to read and fetch the instructions for gameplay. Read smth → pop it. like param.



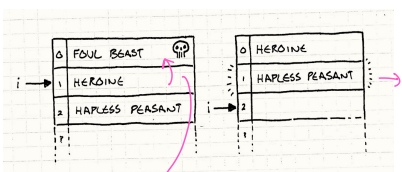
indexing is important.

not go back & forth so many times and copying things so many times, has a memory kind of approach.

* Update Method

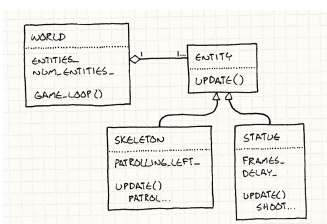
* like a orchestrator

• when to use it?



→ we skipped the peasant without updating it.

Heroine kills the foul beast.



a solution

```
setHealth(0, getHealth(0) + (getAgility(0) + getWisdom(0)) / 2);
```

You might think we'd need instructions to handle the explicit grouping that parentheses give you in the expression here, but the stack supports that implicitly. Here's how you could evaluate this by hand:

1. Get the wizard's current health and remember it.
2. Get the wizard's agility and remember it.
3. Do the same for their wisdom.
4. Get those last two, add them, and remember the result.
5. Divide that by two and remember the result.
6. Recall the wizard's health and add it to that result.
7. Take that result and set the wizard's health to that value.

Do you see all of those "remembers" and "recalls"? Each "remember" corresponds to a push, and the "recalls" are pops. That means we can translate this to bytecode pretty easily.

```
LITERAL 0    [0]              # Wizard index
LITERAL 0    [0, 0]           # Wizard index
GET_HEALTH   [0, 45]          # getHealth()
LITERAL 0    [0, 45, 0]       # Wizard index
GET_AGILITY  [0, 45, 7]       # getAgility()
LITERAL 0    [0, 45, 7, 0]    # Wizard index
GET_WISDOM   [0, 45, 7, 11]   # getWisdom()
ADD          [0, 45, 18]      # Add agility and wisdom
LITERAL 2    [0, 45, 18, 2]   # Divisor
DIVIDE       [0, 45, 9]       # Average agility and wisdom
ADD          [0, 54]          # Add average to current health
SET_HEALTH   []               # Set health to result
```

only for one wizard

actually writing the bytecode??

2 Subclass Sandbox

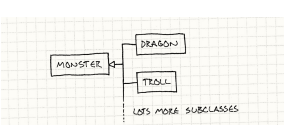
Define behavior in a subclass using a set of operations provided by its base class.

- a base class with lots of derived classes.
- base class provide all methods to derived classes.

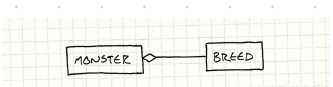
* if all of the derived classes have a common method the base class should surely have this

3 Type Object

Allow the flexible creation of new "classes" by creating a single class, each instance of which represents a different type of object.



inheritance



referenced by

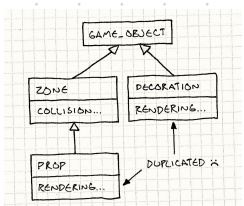
Decoupling Patterns

- Component: Game Object in Unity
- Event Queue
- Service Locator

Why are we need decoupling pattern?

- Easy to adopt
- A change in one does not require a change in another

* Component :
  - Input Component
  - UI Component
  - Physics Component
  - Graphic Component
  - ⋮



* monolitic class

The Pattern

A single entity spans multiple domains. To keep the domains isolated, the code for each is placed in its own component class. The entity is reduced to a simple container of components.

When to Use It

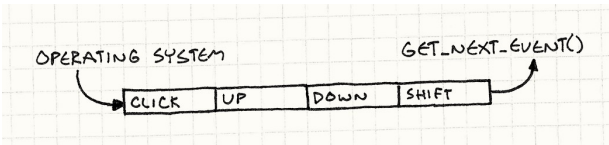
Components are most commonly found within the core class that defines the entities in a game, but they may be useful in other places as well. This pattern can be put to good use when any of these are true:

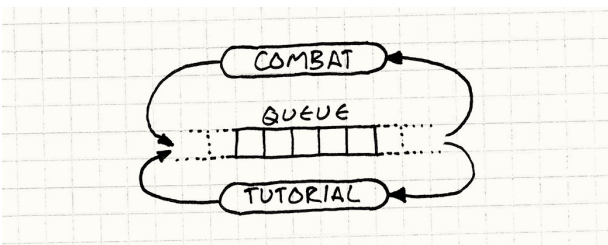
- You have a class that touches multiple domains which you want to keep decoupled from each other.
- A class is getting massive and hard to work with.
- You want to be able to define a variety of objects that share different capabilities, but using inheritance doesn't let you pick the parts you want to reuse precisely enough.

16.05.2024

Event Queue Pattern

Decouple when a message or event is sent from the system.





when to use it?

- play sound

! Queue itself is a global variable, this is a problem.

Things to consider:

- events
- for how long
- messages
- who has access

Service Locator

provides a global point of access to a service without coupling users to the concrete class that implements it.

- random number generators
- logging
- memory allocator

} they are going around everywhere on the database

Final Exam
13 June, 2pm.
II-06

Optimisation Patterns

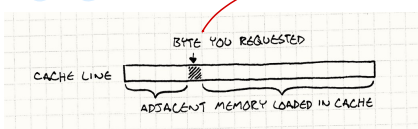
- Data locality
- Dirty flag

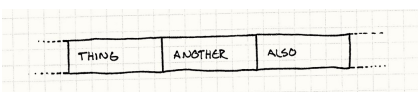
} this week

- Object pool
- Spatial partitioning

} next week

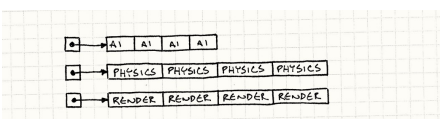
1. Data locality:





→ we will have problems with getting things from memory.

- cache miss



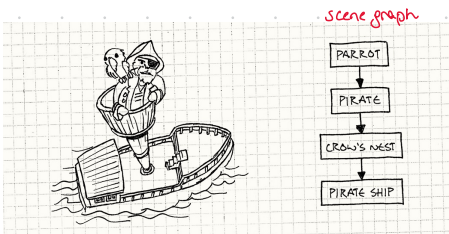
instead of putting everything in a different way, we can organise them in a way that their domain kind of thing.

- we don't have to use it if we cannot organise the data.

23.05.2024

2. Dirty flag:

like true/false



what happens if we kick crow's nest?

⇓

pirate & parrot will be affected

- local & global coordinations are important.

PARROT WORLD = SHIP LOCAL × NEST LOCAL × PIRATE LOCAL × PARROT LOCAL

if we move things:

- → MOVE SHIP
  - RECALC SHIP
    - RECALC NEST
      - RECALC PIRATE
        - RECALC PARROT *
- → MOVE NEST
  - RECALC NEST
    - RECALC PIRATE
      - RECALC PARROT *
- → MOVE PIRATE
  - RECALC PIRATE
    - RECALC PARROT *
- → MOVE PARROT
  - RECALC PARROT *

a "**dirty**" flag tracks when the derived data is out of sync with the primary data.

⇒ where is optimisation?

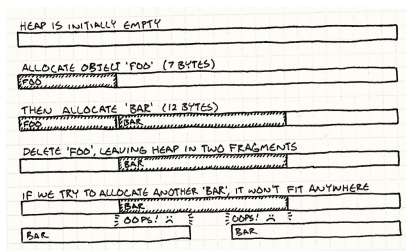
↳ we are not calculating anything before checking the flag first.

3. object pool:

* allocating the fixed amount of memory, if enter change only change what is changed in that memory part.

1. fixed pool of objects

- where to use: visual effects, game physics
- why use?: create very quickly, no memory fragmentation



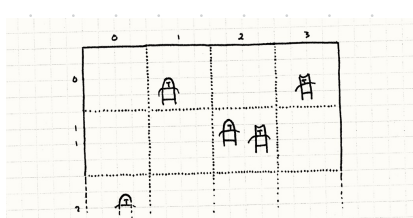
* a collection of reusable objects

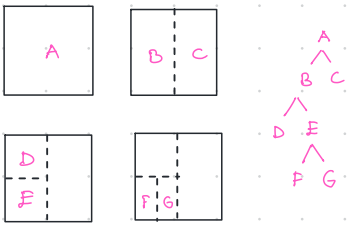
→ keep in mind

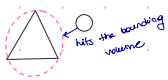
- garbage collector is our job (items are not automatically cleared)
- adjusting the size
- unused object still remain in the memory.

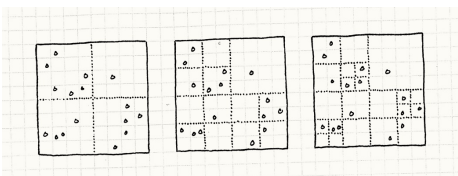
4. Space partition:

space = spatial location









- if objects are not static, this is a problem.
-