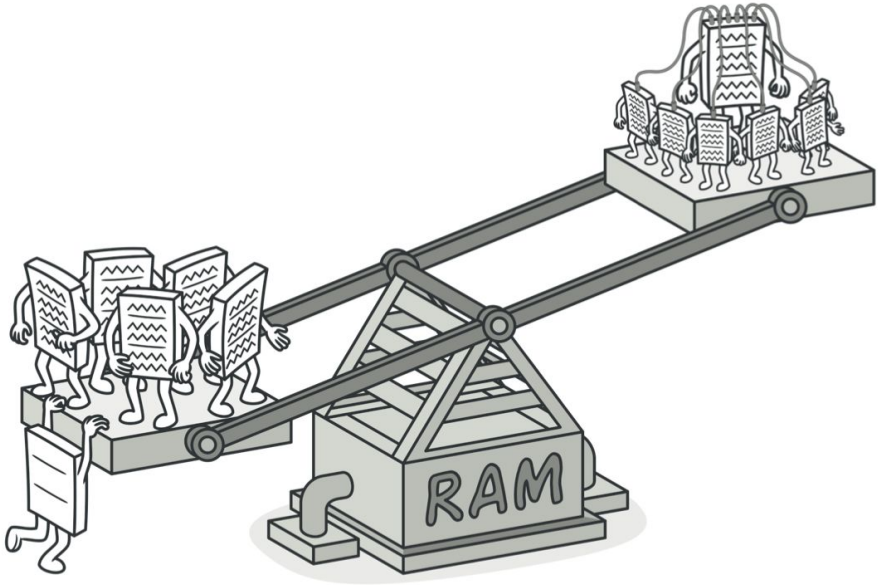
Design Patterns: Flyweight

...

Flyweight

- Structural design pattern
- Lets you fit more objects into the available amount of RAM
- Share common parts of state between multiple objects instead of keeping all of the data in each object.



Flyweight: Problem

- You decide to create a simple video game:
  - players moving around a map and shooting each other.
- You chose to implement a realistic particle system and make it a distinctive feature of the game.
- Vast quantities of bullets, missiles, and shrapnel from explosions should fly all over the map

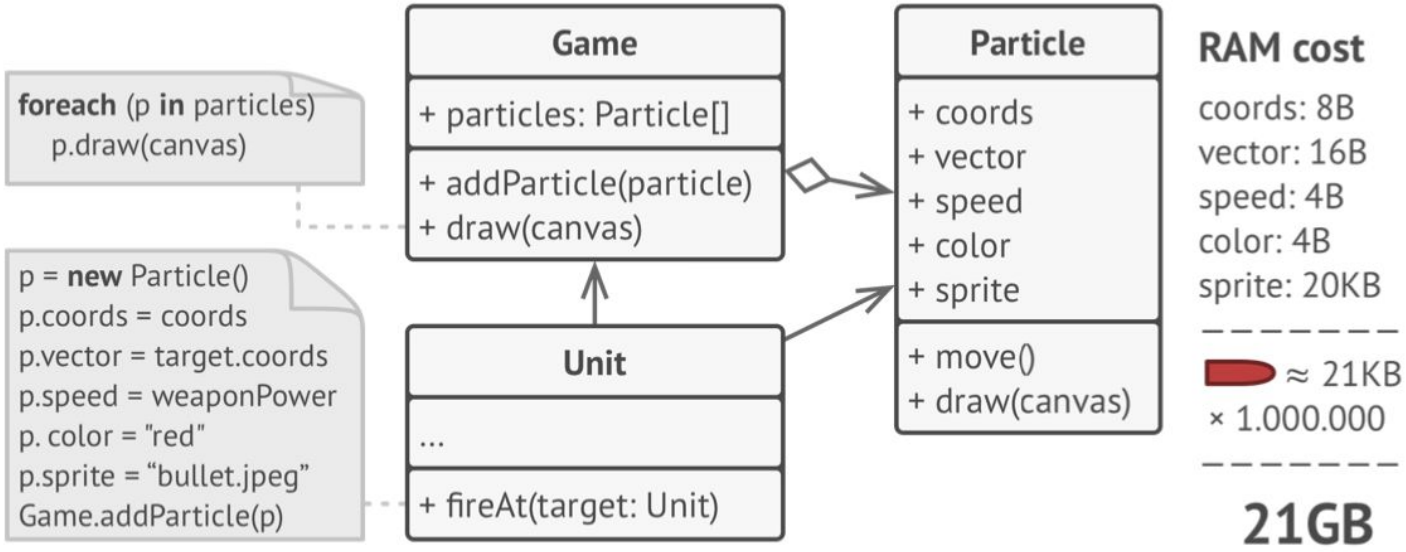
Flyweight: Problem

- Upon its completion, you pushed the last commit, built the game and sent it to your friend for a test drive.
- Although the game was running flawlessly on your machine, your friend wasn't able to play for long.
- On his computer, the game kept crashing after a few minutes of gameplay.

Flyweight: Problem

- After spending several hours digging through debug logs, you discovered that the game crashed because of an insufficient amount of RAM.
- It turned out that your friend's computer was much less powerful than your own computer, and that's why the problem emerged so quickly on his machine.

Flyweight: Problem



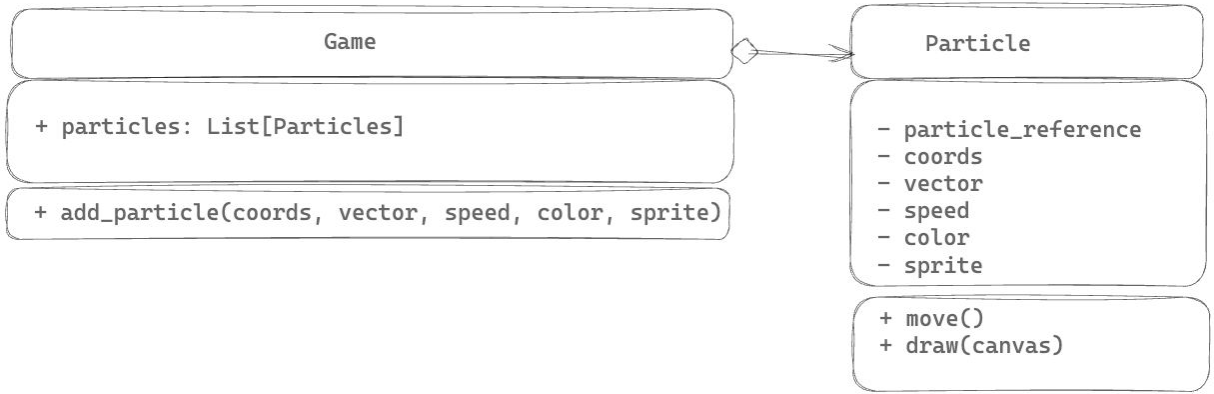
Flyweight: Problem

- The actual problem was related to your particle system.
- Each particle, such as a bullet, a missile or a piece of shrapnel was represented by a separate object containing all related data.
- At some point, newly created particles no longer fit into the remaining RAM, so the program crashed.

Flyweight: Solution

- On closer inspection of the Particle class, you may notice that the **sprite** field consumes a lot more memory than other fields.
- This field stores almost identical data across all particles.
    - For example, all bullets have the same **color** and **sprite**.

Flyweight: Problem



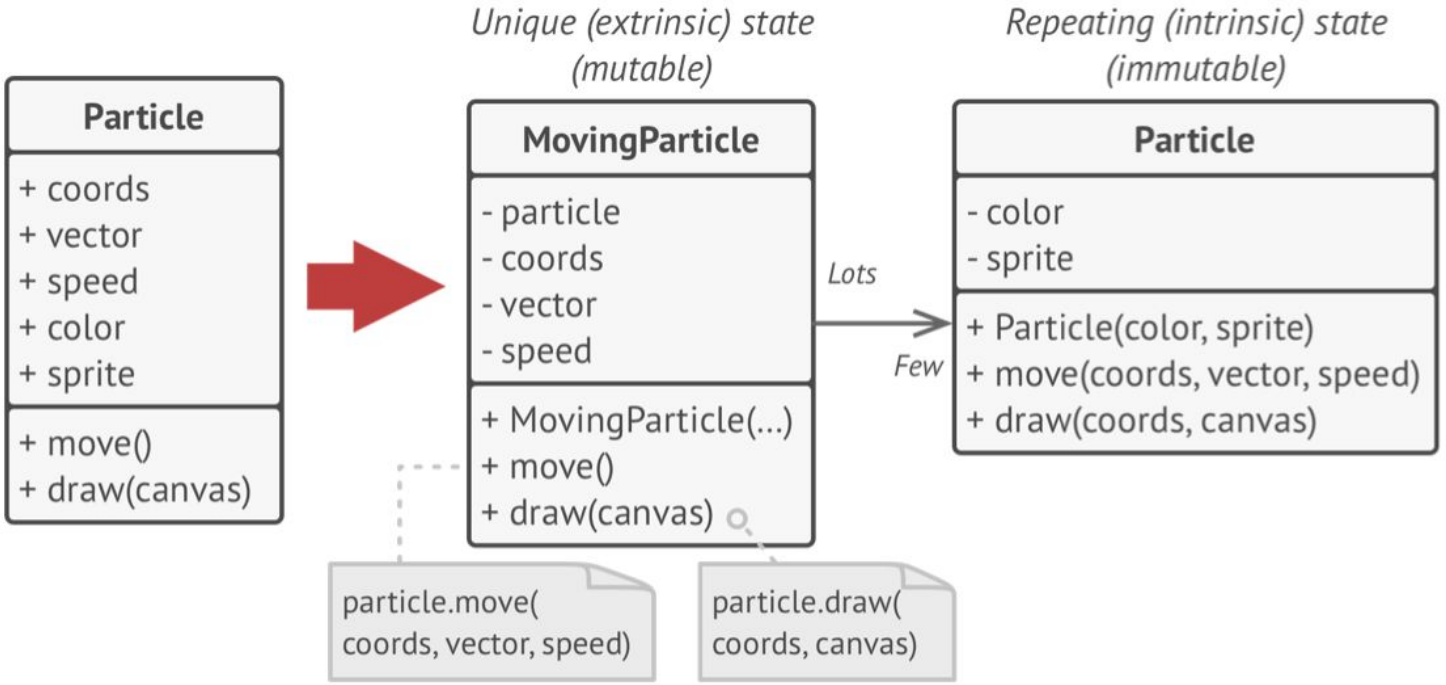
RAM cost

coords: 8B
vector: 16B
speed: 4B
color: 4B
sprite: 20KB

≈ 21KB
× 1.000.000

21GB

Flyweight: Solution



Flyweight: Solution

- Some parts of a particle's state, such as coordinates, movement vector and speed, are unique to each particle.
- The values of these fields change over time.
- This data represents the always changing context in which the particle exists, while the color and sprite remain constant for each particle.

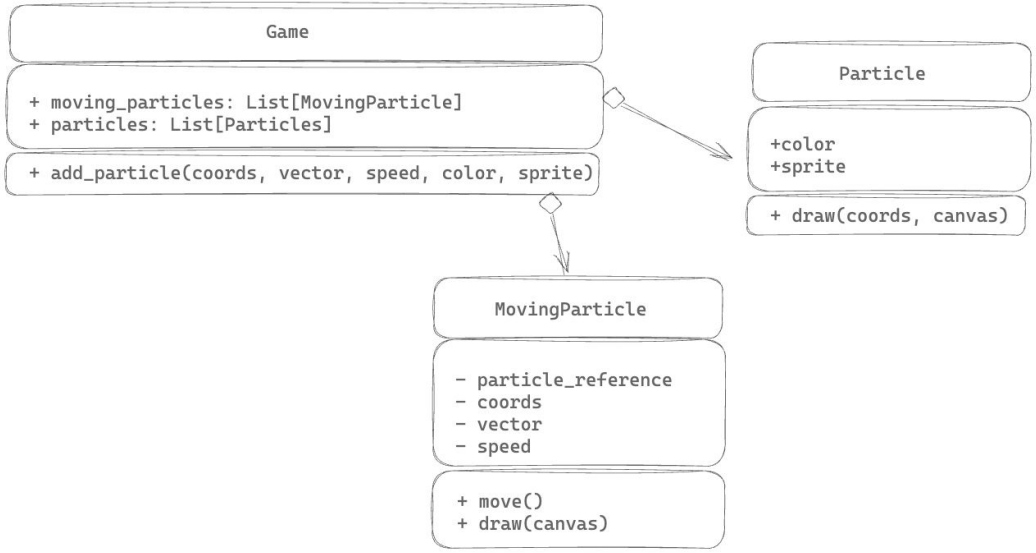
Flyweight: Solution

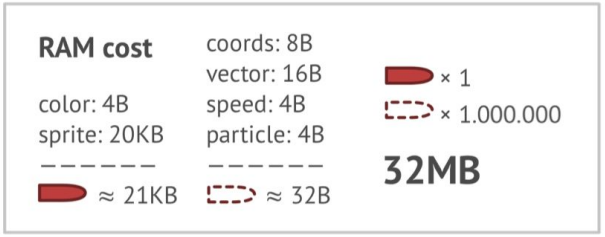
- This constant data of an object is usually called the **intrinsic state.**
- It lives within the object
- Other objects can only read it, not change it
- The rest of the object's state, often altered "from the outside" by other objects, is called the **extrinsic state.**

Flyweight: Solution

- The Flyweight pattern suggests that you store extrinsic state separately from intrinsic state
  - Only the intrinsic state stays within the main object, letting you reuse it in different contexts.
  - Extrinsic state is managed by other classes or functions.
- As a result, you'd need fewer of these large objects since they only differ in the intrinsic state, which has much fewer variations than the extrinsic.

Flyweight: Solution





Flyweight: Solution

- Wait a second! Won't we need to have as many of these contextual objects as we had at the very beginning?
- Technically, yes. But these objects are much smaller than before.
- The most memory-consuming fields have been moved to Particle
  - And there are only a few of those!
- Now, a thousand small contextual objects can reuse a single heavy object
  - Instead of storing a thousand copies of its data.

Flyweight: Immutability

- Since the same Particle object can be used in different contexts by different MovingParticles, you have to make sure that its state can't be modified.

- A flyweight should initialize its state just **once**, via constructor parameters. It shouldn't expose any setters or public fields to other objects.

Flyweight: How to implement

- The Flyweight pattern is merely an optimization.
- Before applying it, make sure your program has a RAM consumption problem related to having a massive number of similar objects in memory at the same time.
- Make sure that this problem can’t be solved in any other meaningful way.

When to use Flyweight

- Use the Flyweight pattern only when your program must support a huge number of objects which barely fit into available RAM.
  - An application needs to spawn a huge number of similar objects
  - this drains all available RAM on a target device
  - the objects contain duplicate states which can be extracted and shared between multiple objects
- Object identity is not important for this part of the application.
- We cannot rely on object identity, because object sharing causes identity comparisons to fail (objects that appear different to the client code end up having the same identity).

Flyweight: Pros and Cons

PRO: You can save lots of RAM, assuming your program has tons of similar objects.

CON: You might be trading RAM over CPU cycles when some of the context data needs to be recalculated each time somebody calls a flyweight method.

CON: The code becomes much more complicated. New team members will always be wondering why the state of an entity was separated.