Question Set: Ex 3 & Ex 4

WHAT IS THE DIFFERENCE BTW A DYNAMIC LINKED LIBRARY & A STATIC LIBRARY?

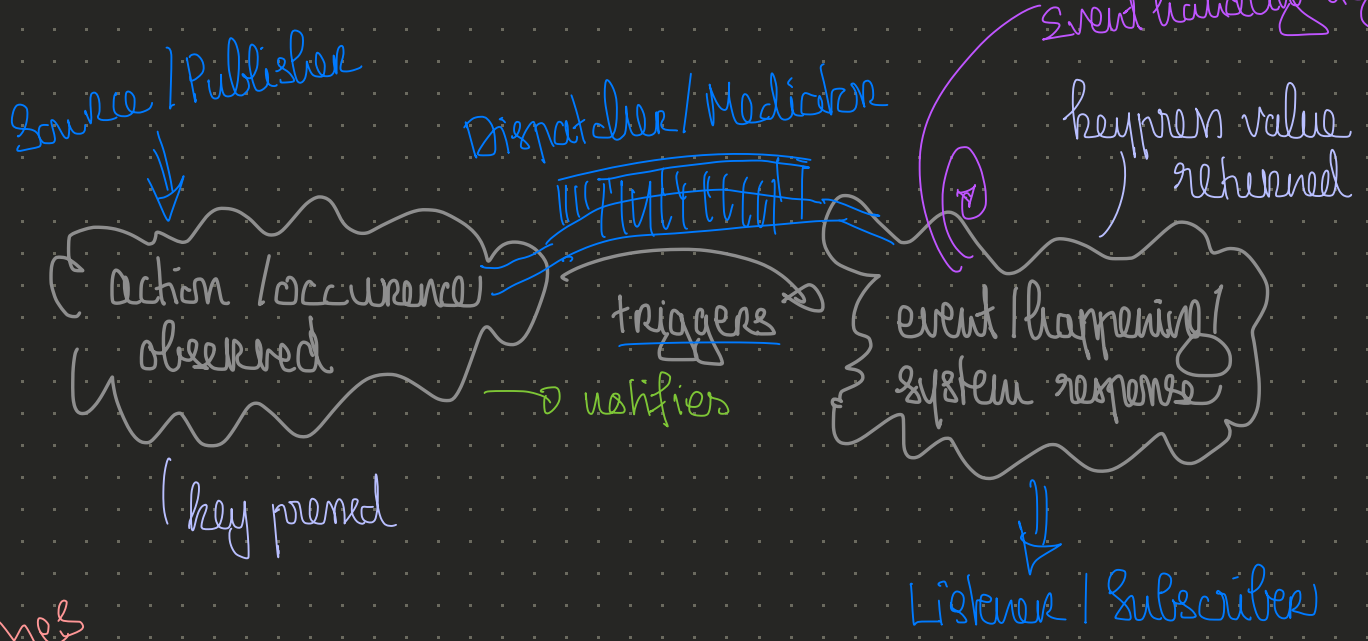
- code from the static library is copied and included directly into the executable during the build process
  - ↳ each programm using the static library getting its own copy of that code
  - ↳ the static library having to be recompiled & relinked when updated
- the executable contains references to the dll
  - ↳ having multiple programms sharing the same library file (loaded into memory & accessible there at runtime)
  - ↳ implementing bugfixes & new features in a new version of the dll without the need for recompilation of the main file

Aspect	Static Library	Dynamic Linked Library
Executable Size	Larger (includes library code)	Smaller (references library code)
Performance	Faster at runtime (no dynamic linking overhead)	Slightly slower (dynamic linking occurs)
Memory Usage	Higher (duplicates library code)	Lower (shared among programs)
Portability	Fully self-contained	Requires external library file
Flexibility	Less flexible (needs recompilation)	More flexible (update library without recompilation)

DESCRIBE AN EVENT SYSTEM IN YOUR OWN WORDS

design pattern which streamlines communication btw. different parts of the system

+ publishers & subscribers do not need to know about each others existence
+ new event types can be added without modifying existing code



priorities

- event dispatching produces extra load on processing
- events cascaded can be difficult to trace
- memory leaks when handled poorly (eg. unsubscription)

Game Engine: can trigger actions on player inputs, collisions or environmental changes

WHAT IS A DESIGN PATTERN ?

approach / structure / workflow to common SW problem set representations

↓

toolkit / best practice

~ draft / template / blueprint of a solution

source: refactoring.guru

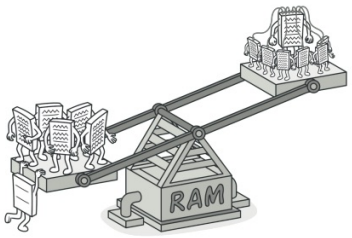
NAME & DESCRIBE 2 DESIGN PATTERNS

Flyweight

Also known as: Cache

Intent

Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

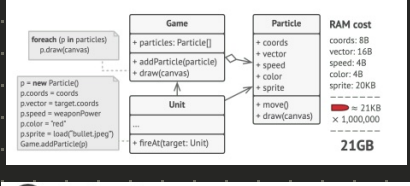


Problem

To have some fun after long working hours, you decided to create a simple video game: players would be 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 and deliver a thrilling experience to the player.

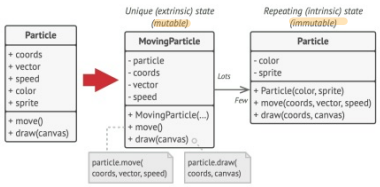
Upon its completion, you pushed the last commit, built the game and sent it to your friend for a test drive. Although the game was running flawlessly on your machine, your friend wasn't able to play for long. On his computer, the game kept crashing after a few minutes of gameplay. After spending several 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 rig was much less powerful than your own computer, and that's why the problem emerged so quickly on his machine.

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 plenty of data. At some point, when the carnage on a player's screen reached its climax, newly created particles no longer fit into the remaining RAM, so the program crashed.



Solution

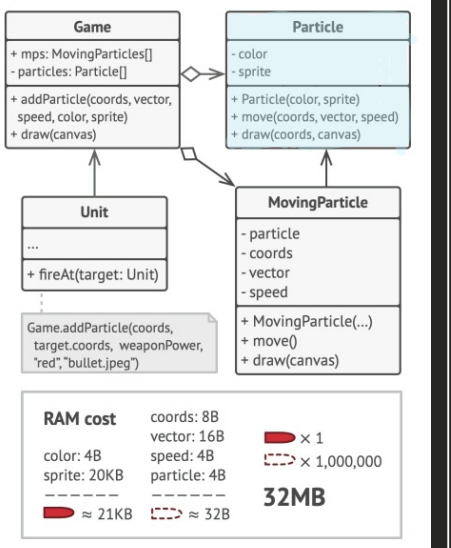
On closer inspection of the `Particle` class, you may notice that the color and sprite fields consume a lot more memory than other fields. What's worse is that these two fields store almost identical data across all particles. For example, all bullets have the same color and sprite.



Other parts of a particle's state, such as coordinates, movement vector and speed, are unique to each particle. After all, 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.

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*.

The Flyweight pattern suggests that you stop storing the extrinsic state inside the object. Instead, you should pass this state to specific methods which rely on it. Only the intrinsic state stays within the object, letting you reuse it in different contexts. As a result, you'd need fewer of these objects since they only differ in the intrinsic state, which has much fewer variations than the extrinsic.



Let's return to our game. Assuming that we had extracted the extrinsic state from our particle class, only three different objects would suffice to represent all particles in the game: a bullet, a missile, and a piece of shrapnel. As you've probably guessed by now, an object that only stores the intrinsic state is called a flyweight.

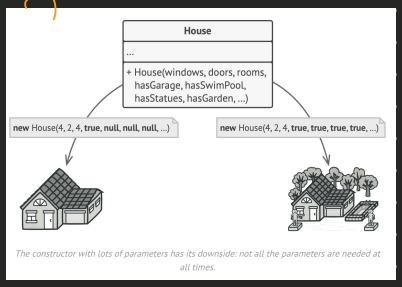
BUILDER

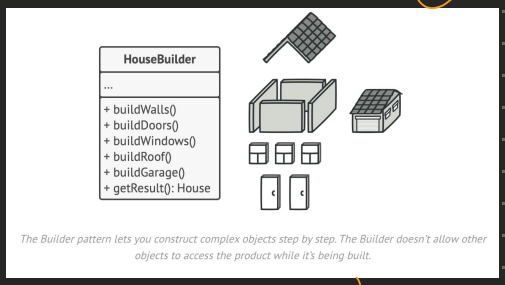
step by step construction of complex objects

↳ "construction code"

eg. many parameters → long & ugly constructor "parameter list"

↳ create subentities = builders that feature their own class & constructions





WHAT IS A PRECOMPILED HEADER? NAME PROS & CONS:

⇒ headers that change infrequently

↓

saved in binary format by the compiler

↓

re-used for subsequent compilations

+ speed up computation

→ no reparsing & processing of the header

Pros of Using Precompiled Headers

1. **Faster Compilation Times:**
   - Parsing headers can be computationally expensive, especially with large libraries or frameworks. Precompiling them reduces redundant work, significantly speeding up builds.
2. **Ease of Use:**
   - Helps developers include a standard set of headers across multiple source files without worrying about duplication overhead.
3. **Consistency:**
   - Encourages the use of a shared set of headers, which can help standardize includes across a project.
4. **Reduced Resource Usage:**
   - Saves CPU time and memory during compilation because the compiler does not repeatedly process the same headers for each translation unit.

Example Workflow

Precompiled Header (`pch.h`):

cpp — Code kopieren

```cpp
// pch.h - Precompiled Header File
#ifndef PCH_H
#define PCH_H

#include <iostream>
#include <vector>
#include <string>
// Add other commonly used headers

#endif
```

Cons of Using Precompiled Headers

1. **Increased Initial Setup Complexity:**
   - Requires careful setup, including selecting appropriate headers for precompilation and ensuring consistent usage across the project.
2. **Dependency on the PCH:**
   - All source files that use the PCH must include the precompiled header as the first include, which can impose restrictions on file structure.
3. **Limited Flexibility:**
   - If a precompiled header changes, the entire PCH must be rebuilt, which can negate the performance benefits for incremental builds.
4. **Potential for Hidden Dependencies:**
   - If a source file implicitly depends on headers included in the PCH, it may break when the PCH is modified or removed, reducing clarity in dependencies.
5. **Incompatibility with Some Build Configurations:**
   - Not all compilers or build systems fully support PCH, and mixing PCH usage with other build techniques can cause issues.
6. **Slower Initial Compilation:**
   - The first compilation to generate the PCH file can be time-consuming, especially with large or complex projects.

...(ChatGPT)...

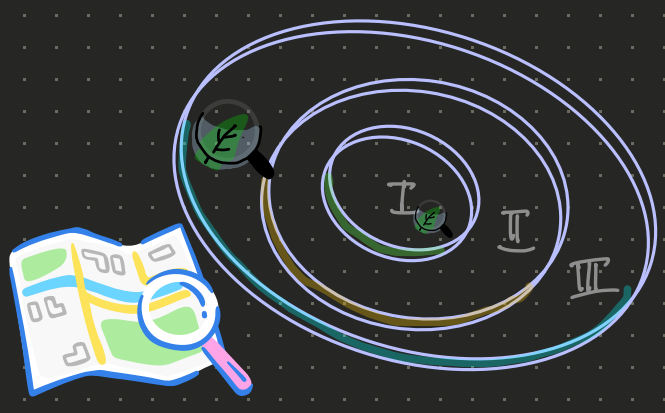
When to Use Precompiled Headers

- Large projects with many translation units and shared dependencies.
- Projects with heavy use of standard libraries or external libraries (e.g., Boost, Qt).
- Development environments where build time is a bottleneck.

When Not to Use Precompiled Headers

- Small projects with few files where the overhead of setting up and maintaining PCH outweighs its benefits.
- Projects with rapidly changing header files, which may trigger frequent PCH rebuilds.

DESCRIBE THE LAYERS OF ABSTRACTION IN YOUR OWN WORDS → vgl. OSI Modell



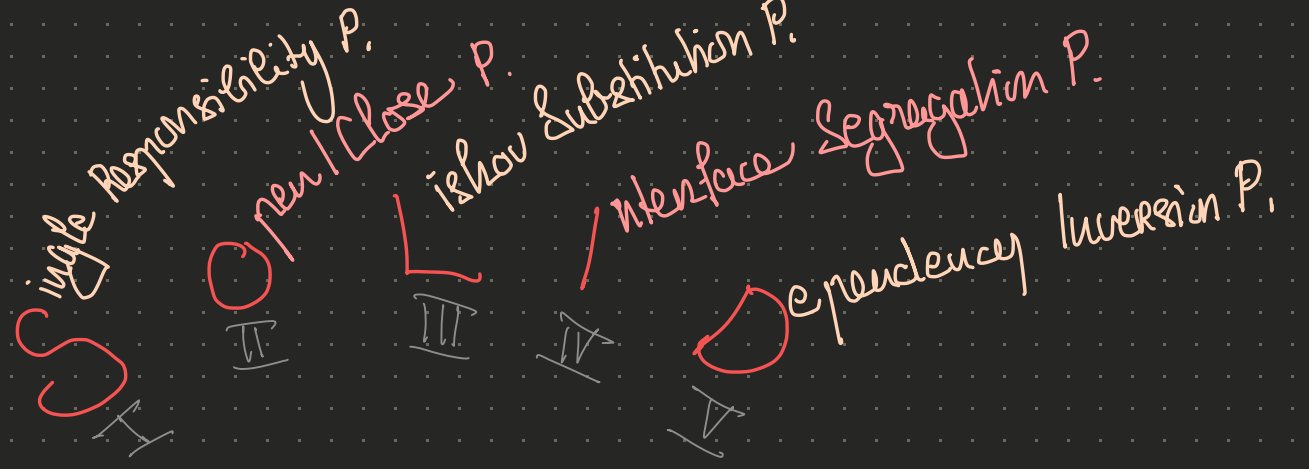
Complexity broken down into chunks

concepts / structures

e.g. physical layer
application layer
transport layer
network layer
data link layer etc.

detail / implementation

NAME THE SOLID PRINCIPLES & DESCRIBE THEM BRIEFLY



I. single piece of functionality per class

II. add functionality without altering existing code ← extendability "Baukästen"

III. subtypes can substitute their base type fully

IV. small specific interfaces rather than large ones

V. reduce coupling btw. high-level & low-level components by the use of interfaces