Game Programming Gems

Design Patterns
↳ Gang of Four

Game Programming Patterns Introduction

Patters are suited for problems in games here:

- Time + Sequencing
- compressed dev time
- interactive elements
- performance is critical

Book Pattern Structure

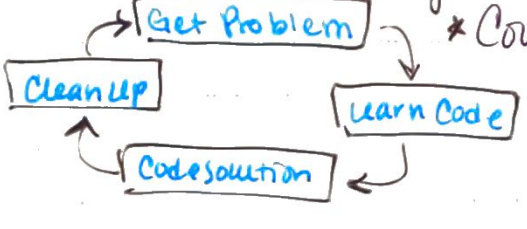
- Intent
- Motivation
- Pattern
- When to Use
- Keep in mind
- Sample Code
- Design Decisions
- See Also

"Each time you use a pattern, you'll likely implement it differently"

Architecture, Performance + Games

↳ how code is organized, when good architecture is one designed for change

* before you make a change you need to understand the code you're changing + the context. Decoupling code speeds up the learning phase by giving you less code you need to learn.



* Coupled code means you need to learn one piece to understand the other. Possibly changing one means changing the other

Key Goal: ↓ amt knowledge needed in-cranium before you can make progress

* ~~Arch~~ Architecture requires time, thought + maintenance. Commonly comes w/ code or language overhead as well.
* Need to think about which parts should be decoupled + abstract them
* but you're always speculating + that's speculation ↑ time, cost + complex. ↳ + Performance
* YAGNI

"It's so easy to get so wrapped up in the code itself that you lose sight of the fact you're trying to ship a game"

* You have to iterate (quickly!) to find the right balance b/w assumptions + flexibility

"One compromise is to keep the code flexible until the design settles down + then tear out some of the abstraction later to improve your performance"

* diff code is good at diff. times, it can be common early on to write code you know you'll throw away. But you HAVE to throw it away.

try writing prototype in a ↘ diff. lang.

Forces in Play

① nice architecture → easier to understand code over time

② Fast runtime performance

③ get today's features done quickly

(aka. all about speed: long-term dev, games exec., short-term dev)

* optimization tends to calcify a code base
* these trade-offs make development EXCITING & DIFFICULT TO MASTER
* try to go for simplicity. get data structures then algorithms right & keep the code base small but readable

"a good solution isn't an accretion of code, it's a distillation of it."

* Don't waste time w/ abstraction & decoupling unless you're confident the code in question needs that flexibility
* Think design for performance but put off the low-level optimizations that calcify your code until as late as possible
* move quickly to explore but not so fast to leave a mess. you have to live w/ it long term
* Don't bother making code you're going to trash pretty

"But, most of all, if you want to make something fun, have fun making it."