Wzorce projektowe – uzupełnienie

Daniel Karaś

Pyłek

Wzorzec pyłek (ang. *flyweight*) zalicza się do strukturalnych wzorców projektowych.

Jego nadrzędnym celem jest zminimalizowanie zużycia pamięci w sytuacji, gdy wymagane jest użycie dużej ilości bardzo podobnych do siebie obiektów.

Pyłek

W publikacji „gangu czworga”, autorzy wyodrębniają w ramach modelowanego konceptu dwa stany:

- **wewnętrzny** (ang. *intrinsic*) – część współdzielona, niezależna od kontekstu.
- **zewnętrzny** (ang. *extrinsic*) – wykorzystuje lub jest wykorzystywany przez stan wewnętrzny do wytworzenia konkretnego kontekstu.

Pyłek

Typowa implementacja tegoż wzorca sprowadza się do stworzenia klasy, która zwraca (lub w przypadku braku wystąpienia - leniwie inicjalizuje) stan wewnętrzny, natomiast klient wykorzystuje go w sobie znanym kontekście na który składa się stan zewnętrzny.

Pyłek – przykład zastosowaia

Jako przykład użycia można łatwo wyobrazić sobie choćby grę strategiczną typu Total War: Shogun 2. Wymaga ona wyświetlania w czasie rzeczywistym tysięcy żołnierzy i gdyby zastosować w jej przypadku podejście naiwne, to szybko wystąpiłyby problemy z ilością wykorzystywanej pamięci.

Pyłek – przykład zastosowania

W tym przypadku można by założyć, że zaabsorbowany ilością jednostek gracz nie będzie zwracał uwagi na detale, którymi mogliby różnić się poszczególni wojowie – dla pewności można też ustawić kamerę wysoko nad ziemią.

Wtedy każdego, dajmy na to, łucznika będziemy mogli opisać przez jego stan wewnętrzny, na który składa się jego model 3D (wierzchołki, tekstury, materiały) i przechowywać tę informację tylko raz.

Pyłek – przykład zastosowania

Gdy zajdzie potrzeba narysowania łucznika pobieramy jego model z „fabryki” i rysujemy w określonej lokalizacji.

Relacje pomiędzy stanem zewnętrznym i stanem wewnętrznym są bardzo elastycznie zdefiniowane i na dobrą sprawę ile implementacji, tyle pomysłów. Dokładniej naświetlam to w komentarzach do kodu źródłowego.

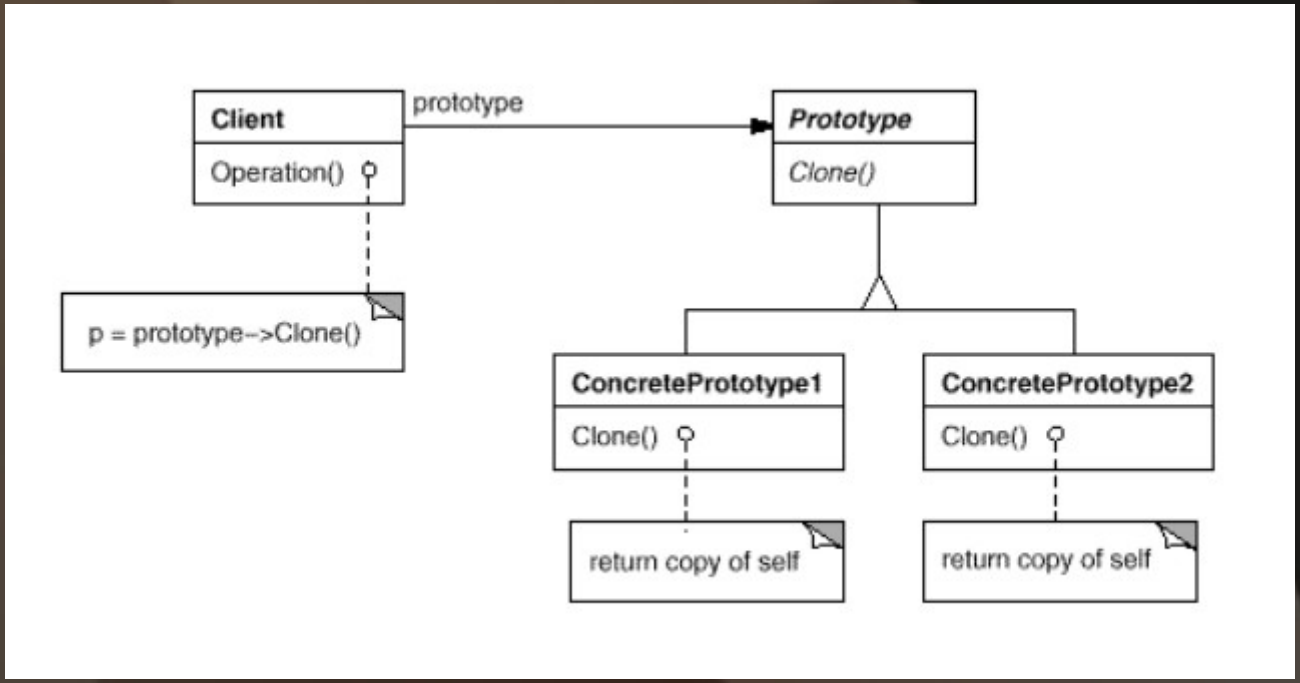
Prototyp

Jest to wzorzec konstrukcyjny, którego filarem jest kopiowanie/klonowanie już utworzonych obiektów.

Jest on przydatny w sytuacjach gdy:

- Konstrukcja obiektów jest kosztowna
- Chcemy ograniczyć ilość klas (w szczególności fabryk), ukryć klasy pochodne przed klientem
- Gdy instancje danej klasy przyjmują tylko określone zestawy stanów, wtedy wygodniej może być przygotować zestaw prototypów i je kopiować

Prototyp


Client
Operation()
prototype
Prototype
Clone()
p = prototype->Clone()
ConcretePrototype1
Clone()
ConcretePrototype2
Clone()
return copy of self
return copy of self

Prototyp

Klient zatem nigdy bezpośrednio nie ma styczności z klasami pochodnymi (ConcretePrototype1, ConcretePrototype 2...).

Z reguły realizuje się to poprzez wprowadzenie managera, który przyjmuje pewien parametr i zwraca odpowiedni prototyp.

Prototyp – przykład zastosowania

Rozważmy (nieco abstrakcyjny) przykład.

Handlujemy pamiątkami w Zakopanem. Dysponujemy głównie ciupagami i drewnianymi szkatułkami, na których dla klienta możemy wystrugać konkretne imię.

Proces wytwarzania ciupag i szkatułek jest czasochłonny, jednak ostatecznie są one do siebie identyczne, z dokładnością do wyrytego na nich potem imienia. Najbardziej optymalnie byłoby więc zrobić jedną ciupagę, jedną szkatułkę, a potem je klonować i takiego klona na życzenie klienta opatrzeć wystruganym imieniem

Prototyp – przykład zastosowania

Z opisu tego łatwo zauważyć, że naszym prototypem będzie pamiątka, natomiast konkretnym prototypem ciupaga i szkatułka.

Teraz wystarczy opatrzyć dwie klasy pochodne w metodę kopiującą i ustawiającą odpowiednie imię dla prototypu.