

2012

Дизайн патерни – просто, як двері

Книга, яка асоціативним та цікавим
способом дозволить вам ознайомитися з
дизайн патернами



DEVELOPER'S
SUCCESS

Андрій Будай



Дизайн патерни – просто, як двері

Або чудні розповіді про патерни

*Книга, яка асоціативним та цікавим способом
дозволить вам ознайомитися з дизайн патернами*

Андрій Будаї

“Design Patterns – easier than simple” by Andriy Buday.

Book which in easy, associative and interesting way introduces GoF Design Patterns.



Згідно із законодавством України, вміст цієї книги є інтелектуальною власністю її автора, тобто мене, Будаї Андрія Івановича. Тому, якщо вам вдасться мати комерційний зиск із матеріалів цієї книги, я, можливо, матиму право судитися із вами в Україні. Поза тим, оскільки нічого нового я не придумав окрім своїх прикладів, а також оскільки основною метою цієї книги є покращення розуміння дизайн патернів серед спільноти програмістів, робота буде поширюватися під ліцензією Creative Commons 3.0 (див. нижче), яка дозволяє вам публікувати, друкувати матеріали із книги, а також модифікувати їх, при умові що використання не має комерційного характеру, а також, що є посилання на автора оригінального тексту.



"Дизайн патерни - просто, як двері" by Andriy Buday is licensed under a Creative Commons Attribution-NonCommercial 3.0 Unported License (<http://creativecommons.org/licenses/by-nc/3.0/>).

Також, якщо українське законодавство зміниться, ліцензія буде перенесена на український аналог. (Приклад дивитись тут: <http://creativecommons.org.ua/переклад-ліцензій>).

«Якщо Ви не можете пояснити щось 6-річній дитині – отже Ви не розумієте цього самі» - Бернард Шоу. На мою думку, автору вдалось навіть більше – йому вдалося знайти досить тонку грань між стисло, доступно та професійно. Тому я би рекомендував цю книгу не тільки студентам чи початківцям, а програмістам які знають, що вчитись чомусь новому ніколи і нікому не пізно. Яскраві приклади, доступні пояснення, достатньо чіткі поради для практичного застосування з професійного досвіду автора, імплементація прикладів у коді – робить книгу дійсно чи не самим доступним україномовним ресурсом з вивчення такої теми дизайн-патернів.

– Роман Мельник

Класна книга по дизайн патернах. Досить цікаве викладення матеріалу. Коли я отримав її екземпляр, то прочитав від початку до кінця не відриваючись. Ідеально підійде початківцям для вивчення основ патернів. Також корисно буде почитати програмістам вищих рівнів, для повторення або просто для фану. Я сам буду по ній повторювати патерни перед евалуейшином.

– Сергій Присяжний

В загальному книгу переглянув, легко читається, дуже добре що код містить форматування із Visual Studio.

– Володимир Мудрик

В цілому дуже радий, що ти нарешті випустив так би мовити бету свого видання, з чим щиро тебе вітаю. Дякую за згадку про мене та мій блог, це чесно кажучи було для мене приємною несподіванкою.

– Ігор Бац

Достатньо хороша книга для початківців, що тільки вступають у світ програмування. Книга легко читається і може стати своєрідним мостом до більш «серйозної» літератури. Дуже добре, що є такі ентузіасти, як автор, що хочуть донести свої знання широкій аудиторії, тим паче українською, що робить книгу ближчою україномовній аудиторії.

– Дмитро Дзюма

блін як гарно виглядає... і написано "від душі"

– Роман Павлов

Зміст

Вступ.....	5
Актуальність книги	5
Із чого усе починалося?	7
Чим ця книга відрізняється від інших подібних?	7
Деякі слова про текст книги	8
Сайт книги та вихідний код.....	8
Подяка	8
Як читати цю книгу?	9
Породжуючі патерни	10
1. Абстрактна Фабрика	11
2. Будівельник.....	14
3. Фабричний Метод	17
4. Прототип	19
5. Одинак (Синглтон).....	24
Структурні патерни.....	27
6. Адаптер.....	28
7. Міст.....	30
8. Компонувальник.....	33
9. Декоратор	36
10. Фасад	39
11. Легковаговик (Флайвейт)	42
12. Проксі	46
Патерни поведінки	51
13. Ланцюжок Відповідальностей	52
14. Команда	55
15. Інтерпретер.....	58
16. Ітератор.....	61
17. Медіатор	64
18. Хранитель (Мemento).....	68
19. Спостерігач	71
20. Стан.....	74
21. Стратегія.....	79
22. Шаблонний метод.....	82
23. Відвідувач	85
Використані матеріали та подальші рекомендації	89
Про автора.....	90

Вступ

- ❖ Актуальність книги
- ❖ Із чого усе починалося?
- ❖ Чим ця книга відрізняється від інших подібних?
- ❖ Деякі слова про текст книги
- ❖ Сайт книги та вихідний код
- ❖ Подяка
- ❖ Як читати цю книгу?

Я дуже радий, що ви взяли цю книгу до рук і проявили бажання прочитати її, або, принаймні, оглянути.

Якщо ви розробник із великим досвідом, користі із цієї книги, швидше за все, ви не отримаєте, тому можете припинити читати, не турбуючи свої нерви та зайвий раз не критикуючи мене. Проте, можливо, деякі із прикладів вам будуть до смаку, а, можливо, ви зможете використати їх для своїх пояснень молодому персоналу, або, скажімо, порекомендуєте початківцям.

Якщо ви тільки починаєте займатися програмуванням професійно, або ж вивчаєте програмування в університеті, ця книга може бути хорошим і цікавим способом ознайомлення із дизайн патернами. А ще вона дозволить легко знаходити аналогії до дизайн патернів під час проходження співбесіди.

Актуальність книги

Почнемо із найжорстокішого – актуальність цієї книги. Можливо ви десь дуже багато чули про те, що кращі програмісти ідеально знають дизайн патерни та ідеально їх застосовують. Це неправда, або принаймні не завжди правда. Є дуже багато програмістів, які, не пригадуючи дизайн патернів, пишуть дуже хороший код. Багато хто просто-напросто не читав про патерни, але із часом здобув певне інтуїтивне відчуття того, як має бути побудований справжній об'єктний код. Моя особиста думка полягає у тому, що в основі доброго розуміння побудови коду, та й дизайну взагалі, лежить принцип єдиної відповідальності¹. Як би там не було, але допоки у програміста не буде цього своєрідного відчуття базових принципів об'єктного програмування, про коректне використання дизайн патернів не може бути й мови. Їхнє розуміння буде нав'язане і однобічне, а використання необдумане та нечітке. Про дизайн патерни можна багато говорити, багато писати, а ви всерівно не будете їх знати, допоки не будете їх використовувати, або краще, допоки не дійдете самостійно до них, винайшовши той же дизайн патерн самостійно.

¹ Single Responsibility - http://en.wikipedia.org/wiki/Single_responsibility_principle

Звичайно що можна прочитати книгу про патерни. Зазвичай усі так і починають. Тут ніби немає нічого поганого, навпаки – структурований академічний підхід. Але є підступність – готовність до прочитання тої чи іншої книги. Для прикладу, часто першокурсникам із курсу програмування C++, пропонують книгу Страуструпа². Молоді студенти, будучи високомотивованими, пристрасно беруться за вивчення C++ і тут їх поглинає вбивча сила думки професора – автора книги. Повторні спроби зрозуміти програмування і C++ видаються примарними і студент втрачає бажання до подальшого вивчення. Звичайно, інша справа, коли таку книгу дають на третьому або четвертому курсі, коли вже є розуміння програмування (скажімо Pascal). Як на мене, давати потужну ґрунтовну книгу на певну тематику можна тільки тоді, коли є базовий досвід по тій тематиці, або коли є постійна практика під наглядом людей, які знають суть справи.

Можна поступити по іншому – почати працювати над проектом, в якому потрібне використання патернів. У такому випадку, ви повинні бути майже генієм із чудовими навиками синтезувати ідеї, або ви матимете помічників більш досвідченіших за вас. Як би там не було, ви себе зловите на тому, що читаєте сторінку на вікіпедії про той чи інший підхід або патерн.

Багато «модних чуваків» багато пишуть і говорять про ті патерни. Я вам скажу (можливо вони також), що не дуже варто заморочуватися тими патернами, бо дуже вірогідно що вам не прийдеться застосовувати їх у сирому вигляді. Таке їхнє застосування це ніщо інше як «притягування за вуха», яке до добра не приводить. Я думаю, що ви вже чули про такі проекти, де понапишали патернів в кожному місці, де хоч трішки пахло проблемою, яка могла б вирішитись якимось із них. В кінці кінців, проект виглядав дуже складно і незрозуміло. Що не становило ніякої проблеми, допоки «крутий чувак» не вирішив змінити місце роботи, а на його місце прийшов початківець. Дивлячись на перенасичений патернами код, більшість програмістів віддають перевагу писати свій код збоку і не вникати в труднощі.

Так виглядає, що, мабуть, була якась хвиля великого піднесення, коли усі вірили, що знайшлася панацея, яка допоможе писати код найправильніше. Таке, мабуть, було роки назад, коли більшість програм були простими клієнт-сервер рішеннями, і не було цієї хмарної розпорошеної сервісно орієнтованої ейфорії³, коли на арену вийшли нові, вищого характеру, проблеми.

²[*The C++ Programming Language*](#) by Bjarne Stroustrup — [Addison-Wesley](#) Pub Co; 3rd edition (February 15, 2000); ISBN 0-201-70073-5

³ Спроба автора пожартувати на новомодні теми Could, distributed systems та SOA.

Патерни – не панацея, і їхнє повсюдне використання також не принесе добра, або навіть може нашкодити. Тому: **Чи актуальна ця книга?**

- Ні. На теперішній час є дуже багато готових рішень у програмуванні, і можна легко оминати застосування 23-ох патернів GoF. Також всесвітня павутина просто завалена прикладами патернів, а досить подібних книг також не бракує.
- Так. Книга дає можливість простим та цікавим способом швидко ознайомитися із патернами, причому на українській мові.

Із чого усе починалося?

Одного чудового зимового дня, а саме 16 січня 2010, я написав блог пост, у якому виразив своє бажання вивчити мову програмування Java. Для цього я мав декілька способів. Одним із яких був мій диплом, що вимагав певних знань у сфері багатопоточності цієї мови, проте мені цього було мало, і я вирішив написати усі дизайн патерни, згадані у книзі «банди чотирьох», на джаві. Цей процес не був дуже жвавим, проте із певного моменту я почав вести рубрику «патерн вівторка» на .NET User Group Львова. Для цієї рубрики я був змушений: а) перекладати і б) перекладати. У першому випадку то був переклад із англійської на українську, а у другому із Java на C#. Коли у першому випадку я був змушений замислюватись над перефразуванням речень, щоб вони звучали менш-більш нормально, то у другому це був суцільний «копі-пейст». Таким чином я зробив для себе висновок, що іншу мову програмування слід вчити, але треба бути певним у способі, яким ви це будете робити.

У іншому блог пості я обмовився, що все ж таки наважився писати свою першу книгу. Звичайно, я знав, що вона не може бути повноцінною, видатною авторською книгою. Проте, з іншої сторони, чи ви коли небузь задумувалися над тим, щоб написати свою книгу, навіть маленьку? Якщо так, то чи є вона зараз написана? Якщо ні, то варто над чимось задуматися. Я не хотів задумуватися, я просто вирішив, що треба із чогось починати. Ось ця книга і буде моєю першою спробою.

Чим ця книга відрізняється від інших подібних?

В тому ж блог пості, я описав, чим книга буде відрізнятися від інших її подібних книг:

1. Вона не буде перекладом ориганальної книги.
2. Вона буде містити унікальні приклади, придумані мною.
3. Вона буде написана доступною і простою мовою.
4. Вона буде безплатною електронною книгою.
5. Вона буде на українській мові.

Деякі слова про текст книги

Із самого початку книга вам не видасться написаною чистою літературною мовою. Книга може містити програмістський жаргон та англійські слова. Особисто я вважаю це більш правильним способом написати легку до читання книгу для такої специфічної аудиторії як програмісти, а також, враховуюче те, що книга буде поширюватися в електронному варіанті.

В книзі також використовуються різного типу форматування, зокрема класи та методи зображаються *ThisWay*, назви патернів зображаються *Прототип*, якщо назва патерну не дуже поширена, то інша назва буде вказана в дужках. Уривки коду також мають специфічні позначення. Код форматований так, як він був би відформатований та забарвлений у Visual Studio.

Сайт книги та вихідний код

Я створив невеличкий веб-сайт для цієї книги, який доступний за адресою <http://designpatterns.andriybuday.com/>.

Кожен із прикладів у книзі має реалізацію на мові програмування C#. Завантажити всі сорси можна за на [спеціальній сторінці сайту](#).

Подяка

Дуже часто книга не пишеться однією людиною, особливо на обширні технічні теми. Так вийшло, що ця книга є одноавторською, але звичайно, що не обійшлося без допомоги інших людей. В першу чергу хочу подякувати своїй дружині Наталії, не тільки за моральну підтримку (зазвичай тільки за це дякують автори книг, хоча вона не є маловажною), але й за допомогу у корекції та форматуванні книги.

Також хочу подякувати всім, хто читали мої блог пости про дизайн патерни, як на моєму блозі, так і на блозі львівської юзер групи. Зокрема подяка таким людям: Присяжний Сергій, Накрийко Андрій, SVGre⁴, Bats Ihor⁵, Andrii, Дзюма Дмитро (DixonD)⁶, Andrew Zak, leonidv, IhorCo, Andy, Kalita Roman, ufo_133, DrAlligieri, Oleh Sklyarenko, Genrih, Yuriy Seniuk⁷, Yevhen Bobrov⁸, Atski⁹, Petro, Геннадій Омельченко, Андрій. Імена (або ніки) ніяк не посортовані і взяті із коментарів на блог постах.

⁴ <http://svgreg.blogspot.com/>

⁵ <http://batsihor.blogspot.com/>

⁶ <http://dixonD.blogspot.com/>

⁷ <http://yuriyseniuk.blogspot.com/>

⁸ <http://blog.xtalion.com/>

⁹ <http://tskitishvili.blogspot.com/>

Особлива подяка Присяжному Сергію, Бац Ігорю та Накрийко Андрію як найбільш активним читачам моїх постів про дизайн патерни.

Дуже дякую рецензентам книги, а саме: Бац Ігор, Присяжний Сергій, Дзюма Дмитро, Мельник Роман, Захарко Андрій, Мудрик Володимир і великій кількості інших людей, які читали «бета-версію» книги, проте не надіслали своїх думок.

Також хочу подякувати компанії «СофтСерв» за ті незабутні роки, які я провів, працюючи там із чудовими людьми, особливо у команді Mobile.

Як читати цю книгу?

Якщо ця книга є першою, по якій ви збираєтесь вивчати дизайн патерни, почніть читати її у такому порядку: *Фасад*, *Медіатор*, *Одинак*, *Будівельник*, *Шаблонний Метод*, *Декоратор* а далі переключайтесь на інші. Такі дизайн патерни, як *Абстрактна Фабрика*, *Компонувальник*, *Відвідувач*, *Інтерпретер* раджу відкласти на потім, оскільки вони не є дуже легкими для сприйняття.¹⁰

Оскільки за основу була взята оригінальна книга про дизайн патерни, а також, оскільки, суть тієї книги була у структуризації дизайн патернів, було б недобре ухилитись від стандартного впорядкування книги. Тому книга містить три розділи із 23-ома дизайн патернами. Кожен із дописів про дизайн патерн не має дуже чіткої структуризації, як у GoF книзі, але має свою дещо змінену структуризацію: спочатку віддалений вступ, який описує проблему, зазвичай дуже віддалену від програмування, а потім ця проблема вирішується за допомогою патерну, причому за допомогою вибудовування асоціативних зв'язків між складовими частинами патерну та проблеми, згаданої у прикладі. Після цього наводиться програмне рішення на мові програмування C#, а також деякі суміжні роз'яснення. Інколи вони пропущені, щоб змусити читача уважно читати код. Під кінець багатьох із патернів наводиться UML-діаграма програмного рішення, причому вона може мати відмінності від діаграм наведених в оригінальній книзі, оскільки це UML-діаграма наведеного прикладу.

Книгу, звичайно, можна читати у відповідності до свого настрою, перестрибуючи від одного патерна до іншого, оскільки немає ніяких чітких залежностей між ними.

Цікавого читання!

¹⁰ <http://mahemoff.com/paper/software/learningGoFPatterns/>

ПОРОДЖУЮЧІ ПАТЕРНИ

Так уже прийнято, що усі дизайн патерни поділені на три великі групи, а саме: *породжуючі*, *структурні* та *поведінкові*. Звичайно, що можна було б опустити вступ до кожної із груп, але, насправді, поділ на групи має досить велике значення. Не даремно хлопці із «банди чотирьох» вибрали саме 23 патерни а не більше і не менше, і не даремно вони поділили їх на ці групи. Це ж була основна мета їхньої роботи – структурувати та формалізувати вже існуючі дизайн патерни.

Отже, породжуючі патерни. Основним завданням таких патернів є спростити створення об'єктів, які необхідні аплікації.

Інколи ви працюєте із певним набором об'єктів через групу інтерфейсів. А тоді хочете створювати об'єкти тільки із іншого набору, щоб пристосувати ваш код до інших умов. Звичайно група інтерфейсів, через які ви оперуєте, залишається та ж сама. Спростити створення відповідного набору допоможе *Абстрактна Фабрика*.

А інколи структура деякого об'єкта дуже складна і залежить від багатьох чинників. Щоб спростити створення такого об'єкту зазвичай використовують *Будівельника*.

А щоб зручно вибрати одну реалізацію та інстанціювати її, відштовхуючись від простої умови, можна використати *Фабричний Метод*.

Нерідко постає завдання отримати копію уже існуючого об'єкта, або отримати можливість швидко генерувати багато подібних екземплярів. У такому випадку *Прототип* якраз пригодиться.

Вибаглеве множення об'єктів не єдине завдання, яке вам слід буде виконувати у роботі, вам часто буде потрібно робити все точно навпаки – мати один-єдиний екземпляр об'єкта і ні за яких обставин не доступатися до йому подібних. Функціональність єдиного екземпляра забезпечуються *Синглтоном*.

1. Абстрактна Фабрика



Уявімо, що ви прийшли в іграшковий магазин (відіграючи роль діда Мороза¹¹) і хочете купити іграшок дітям (і не обов'язково своїм). Мартуся любить плюшеві іграшки, вона часто із ними лягає у ліжко спати. А Дмитрик страшний розбишака, ламає все на світі, рве м'які іграшки і, зазвичай, віддає перевагу гратися із твердими, дерев'яними іграшками. Двоє дітей хочуть ведмедика і котика і ще купу інших тваринок. На щастя, магазин має широкий асортимент забавок і вам вдалося вдосталь закупитися. В один мішок ви накидали дерев'яних іграшок, а в інший плюшевих.

Таким чином, коли ви підійшли до Мартусі, яка любить м'які іграшки, ви витягали із свого мішка спочатку плюшевого ведмедя, а далі плюшевого котика і так далі. Аналогічно ви підійшли до Дмитрика і подарували йому дерев'яного ведмедика і котика, і собаку, і слона, і бегетота... і крокодила...

Абстрактна фабрика надає простий інтерфейс для створення об'єктів, які належать до того чи іншого сімейства.¹²

В нашому прикладі сімейством є набір іграшок-тварин, які по-сімейному реалізують базові класи ведмедика (*Bear*), kota (*Cat*), і інші. Тобто повний звіринець певної реалізації, дерев'яної або плюшевої, і буде сімейством. Конкретною фабрикою є мішок. Одна із конкретних фабрик повертає дерев'яні іграшки, а інша повертає плюшеві. Тому якщо одна дитина просить котика, то їй вернуть реалізацію котика у відповідності до інстанційованого мішка із іграшками.

Я сподіваюся, що приклад із аналогіями не видався заплутаним. А якщо все ж таки видався, пропоную подивитися трохи коду. *Абстрактна фабрика* визначає інтерфейс, що повертає об'єкти kota або ведмедя (базові класи). Конкретні реалізації фабрики повертають конкретні реалізації іграшок потрібного сімейства.

Уривок коду 1.1. Інтерфейс абстрактної фабрики та дві конкретні реалізації

```
// абстрактна фабрика (abstract factory)
public interface IToyFactory
{
    Bear GetBear();
    Cat GetCat();
}
```

¹¹ Життя складається із 4 фаз: а) Ви вірите в Діда Мороза (Миколая/Санту). б) Ви більше не вірите. в) Ви самі Дід Мороз. г) Ви схожі на Діда Мороза.

¹² **Abstract Factory.** Intent: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Абстрактна фабрика. Призначення: Надати інтерфейс для створення сімейств споріднених або залежних об'єктів без зазначення їхніх конкретних класів.

```
// конкретна фабрика (concrete factory)
public class TeddyToysFactory : IToyFactory
{
    public Bear GetBear()
    {
        return new TeddyBear();
    }
    public Cat GetCat()
    {
        return new TeddyCat();
    }
}
// і ще одна конкретна фабрика
public class WoodenToysFactory : IToyFactory
{
    public Bear GetBear()
    {
        return new WoodenBear();
    }
    public Cat GetCat()
    {
        return new WoodenCat();
    }
}
```

Уже зрозуміло, що як тільки ми маємо якийсь екземпляр фабрики, ми можемо плодити сімейство потрібних іграшок. Тому глянемо на використання:

Уривок коду 2.2. Використання конкретної фабрики для дерев'яних іграшок

```
// Спочатку створимо «дерев'яну» фабрику
IToyFactory toyFactory = new WoodenToysFactory();
Bear bear = toyFactory.GetBear();
Cat cat = toyFactory.GetCat();
Console.WriteLine("I've got {0} and {1}", bear.Name, cat.Name);
// Вивід на консоль буде: [I've got Wooden Bear and Wooden Cat]
```

Уривок коду 3.3. Використання конкретної фабрики для плюшевих іграшок

```
// А тепер створимо «плюшеву» фабрику, наступна лінійка є єдиною різницею в коді
IToyFactory toyFactory = new TeddyToysFactory();
// Як бачимо код нижче не відрізняється від наведеного вище
Bear bear = toyFactory.GetBear();
Cat cat = toyFactory.GetCat();
Console.WriteLine("I've got {0} and {1}", bear.Name, cat.Name);
// А вивід на консоль буде інший: [I've got Teddy Bear and Teddy Cat]
```

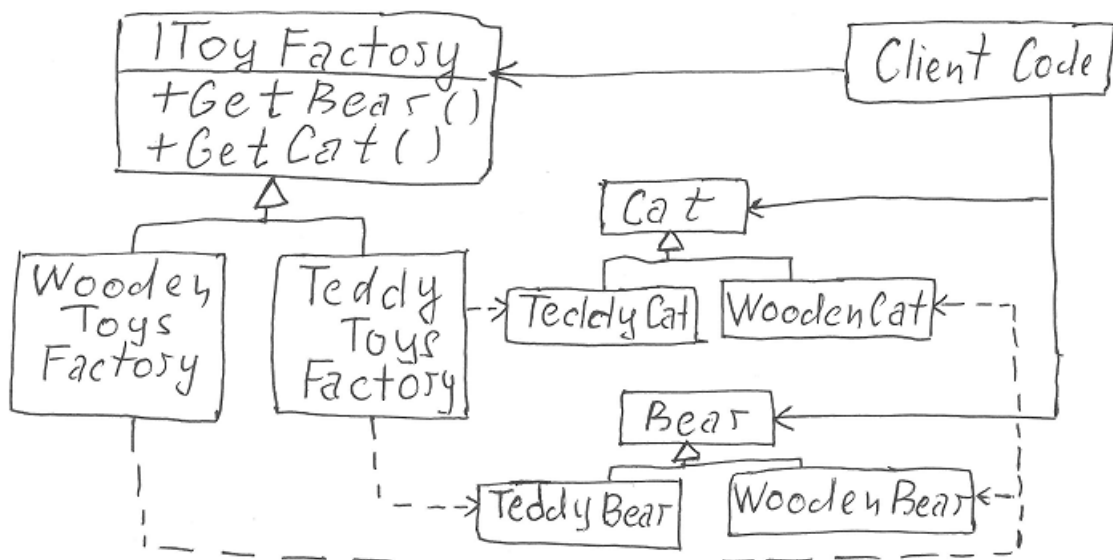
Два куски коду майже абсолютно ідентичні – різниця тільки в конкретному «мішку». Якщо вас ще цікавлять *реалізації іграшок-тваринок*, то вони доволі тривіальні.

Уривок коду 1.3. Реалізації іграшок-тваринок

```
// Базовий клас для усіх котиків, базовий клас AnimalToy містить Name
public abstract class Cat : AnimalToy
{
    protected Cat(string name) : base(name) { }
}
```

```
// Базовий клас для усіх ведмедиків
public abstract class Bear : AnimalToy
{
    protected Bear(string name) : base(name) { }
}
// Конкретні реалізації
class WoodenCat : Cat
{
    public WoodenCat() : base("Wooden Cat") { }
}
class TeddyCat : Cat
{
    public TeddyCat() : base("Teddy Cat") { }
}
class WoodenBear : Bear
{
    public WoodenBear() : base("Wooden Bear") { }
}
class TeddyBear : Bear
{
    public TeddyBear() : base("Teddy Bear") { }
}
```

Абстрактна Фабрика є дуже широковикористовуваним дизайн патерном. Дуже яскравим прикладом буде ADO.NET *DbProviderFactory*¹³, яка є абстрактною фабрикою, що визначає інтерфейси для отримання *DbCommand*, *DbConnection*, *DbParameter* і так далі. Конкретна фабрика *SqlClientFactory* поверне відповідно *SqlCommand*, *SqlConnection* і так далі. Що дозволяє працювати із різними джерелами даних. Дякую за те, що дочитали про цей дизайн патерн саме із моєї книги та із моїм прикладом.



UML-діаграма 1. Абстрактна фабрика

¹³ DbProviderFactory Class: [http://msdn.microsoft.com/en-us/library/system.data.common.dbproviderfactory\(v=vs.80\).aspx](http://msdn.microsoft.com/en-us/library/system.data.common.dbproviderfactory(v=vs.80).aspx)

2. Будівельник



Уявіть, що ви володієте магазином (гаражем) з продажу персональних комп'ютерів, в якому можна вибирати конфігурацію прямо біля каси (як піцу в піцерії). Вам слід створити систему, що дозволить легко *будувати* будь-яку конфігурацію ноутбука для будь-якого покупця. Причому стандартні конфігурації мають складатися «по накату». От як піца може бути «фірмова», «грибна», «справжня італійська», «пепероні», або ваша специфічна, так і ноутбук: для геймера або для подорожей. Влаштуємо бізнес?

Будівельник вимальовує стандартний процес створення складного об'єкта, розділяючи логіку будування об'єкта від його представлення.¹⁴

Шляхом додавання певних частин, таких як процесор, пам'ять, жорсткий диск, батарея і т.д, можна скласти повноцінний комп'ютер. Часто конфігурації можуть набути певних стандартів. В кінці кінців ваш продавець балакає із клієнтом і запитує, яку пам'ять він хоче, і так далі. Або ж, якщо клієнт не дуже «шаруючий», продавець може запитати: «Ну вам той комп треба для шпільок? Яких?». Звісно, що ви наперед знаєте певні кроки, щоб створити ноутбук. Ці кроки визначаються в *абстрактному будівельнику* (*abstract builder*). Єдине що вам потрібно від клієнта, це дізнатися, які компоненти використовувати на кожному кроці.

Уривок коду 2.1. Абстрактний будівельник

```
abstract class LaptopBuilder
{
    protected Laptop Laptop { get; private set; }
    public void CreateNewLaptop()
    {
        Laptop = new Laptop();
    }
    // Метод, який повертає готовий ноутбук назовні
    public Laptop GetMyLaptop()
    {
        return Laptop;
    }
    // Кроки, необхідні щоб створити ноутбук
    public abstract void SetMonitorResolution();
    public abstract void SetProcessor();
    public abstract void SetMemory();
    public abstract void SetHDD();
    public abstract void SetBattery();
}
```

¹⁴ **Builder.** Intent. Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Будівельник. Призначення. Розділити створення складного об'єкта від його представлення, щоб той же процес створення міг утворити різні представлення.

Якщо покупець відповідає «Ух, так! Я хочу шпіляти... ууеех!» і у нього загорілися очі, то ви вже наготові і маєте конкретну реалізацію для ігрового ноутбука:

Уривок коду 2.2. Конкретна реалізація будівельника

```
// Таким будівельником може бути працівник, що
// спеціалізується у складанні «геймерських» ноутів
class GamingLaptopBuilder : LaptopBuilder
{
    public override void SetMonitorResolution()
    {
        Laptop.MonitorResolution = "1900X1200";
    }
    public override void SetProcessor()
    {
        Laptop.Processor = "Core 2 Duo, 3.2 GHz";
    }
    public override void SetMemory()
    {
        Laptop.Memory = "6144 Mb";
    }
    public override void SetHDD()
    {
        Laptop.HDD = "500 Gb";
    }
    public override void SetBattery()
    {
        Laptop.Battery = "6 lbs";
    }
}
```

Але якщо ваш покупець бізнесмен і переглядає презентації і звіти, перелітаючи через атлантику:

Уривок коду 2.3. Конкретний будівельник для ноутбуків

```
// А ось інший «збирач» ноутів
class TripLaptopBuilder : LaptopBuilder
{
    public override void SetMonitorResolution()
    {
        Laptop.MonitorResolution = "1200X800";
    }
    public override void SetProcessor()
    {
        //.. і так далі...
    }
}
```

Для того, щоб справитися із побудовою ноутбука, базуючись на відповіді покупця, знадобиться директор (*director*):

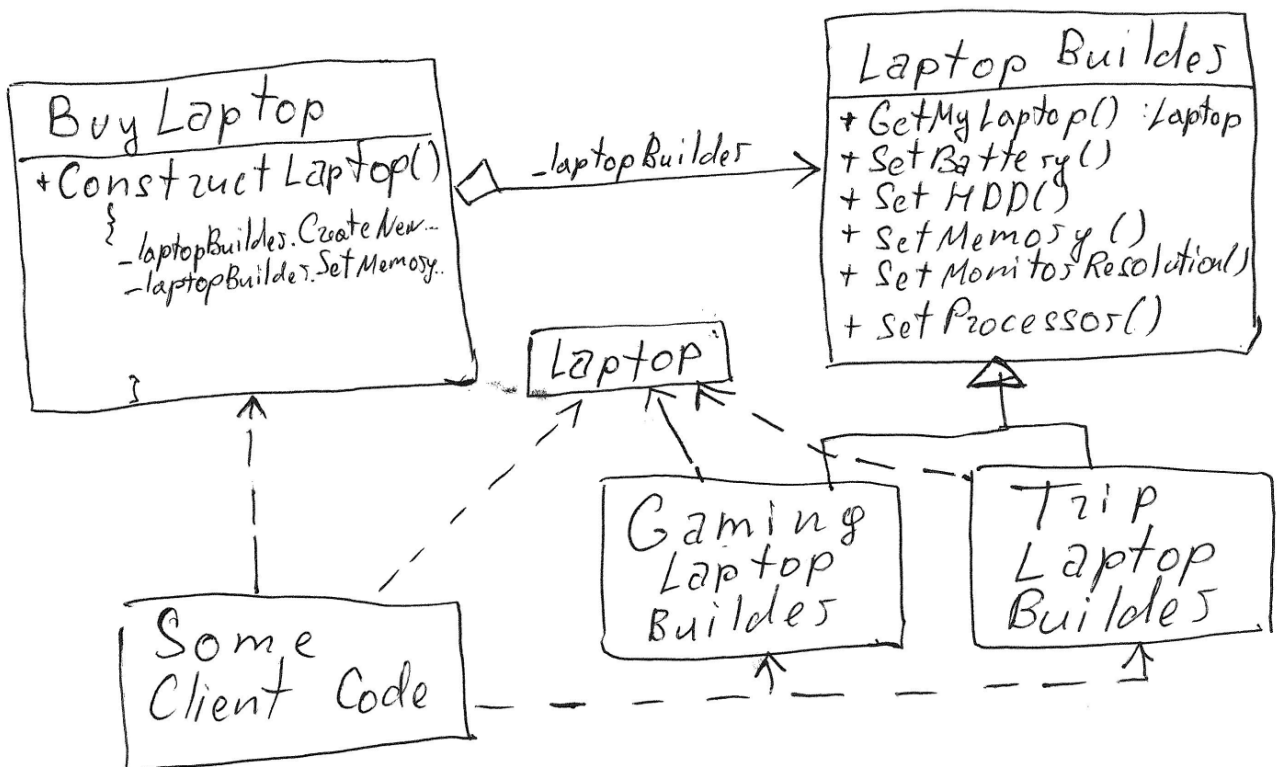
Уривок коду 2.4. Директором може бути код приймаючий замовлення

```
class BuyLaptop
{
    private LaptopBuilder _laptopBuilder;
    public void SetLaptopBuilder(LaptopBuilder lBuilder)
    {
        _laptopBuilder = lBuilder;
    }
}
```

```
// Змушує будівельника повернути цілий ноутбук
public Laptop GetLaptop()
{
    return _laptopBuilder.GetMyLaptop();
}
// Змушує будівельника додавати деталі
public void ConstructLaptop()
{
    _laptopBuilder.CreateNewLaptop();
    _laptopBuilder.SetMonitorResolution();
    _laptopBuilder.SetProcessor();
    _laptopBuilder.SetMemory();
    _laptopBuilder.SetHDD();
    _laptopBuilder.SetBattery();
}
}
```

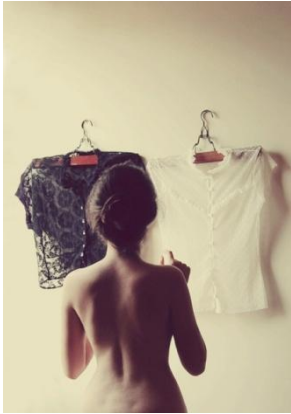
Уривок коду 2.5. А тепер глянемо на використання цього патерну

```
// Ваша система може мати багато конкретних будівельників
var tripBuilder = new TripLaptopBuilder();
var gamingBuilder = new GamingLaptopBuilder();
var shopForYou = new BuyLaptop(); // Директор
// Покупець каже, що хоче грати ігри
shopForYou.SetLaptopBuilder(gamingBuilder);
shopForYou.ConstructLaptop();
// Ну то нехай бере що хоче!
Laptop laptop = shopForYou.GetLaptop();
Console.WriteLine(laptop.ToString());
// Вивід: [Laptop: 1900X1200, Core 2 Duo, 3.2 GHz, 6144 Mb, 500 Gb, 6 lbs]
```



UML-діаграма 2. Будівельник

3. Фабричний Метод



Уявіть, що ваша аплікація є дуже складною, і так склалося, що ви використовуєте два логінг провайдери: один *Log4Net* та інший *Enterprise.Logging*. Ваш колега додумався помістити вибір провайдера прямо у конфігураційний файл. Так як ви всю логіку логування абстрагуєте за інтерфейсом *ILogger*, то вам не хотілося б, щоб при потребі логгера вам приходилося по умові перевіряти що записано у конфізі і тоді створювати необхідний екземпляр. Мабуть, було б добре приховати специфіку створення конкретного провайдера та винести її в окремий клас. Скажімо в *Фабричний Метод*.

Фабричний Метод вирішує яку реалізацію інстанціювати. Вирішують або нащадки фабричного методу, або він сам, приймаючи якийсь параметер.¹⁵

Як на мене, то цей дизайн патерн є одним із найбільш відомих і найпростіших. Я переконаний, що більшість читачів бачили його багато раз. Завдання *Фабричного Методу* полягає в прихованні конкретного класу, що має бути створений та повернений під виглядом спільної абстракції. Якщо в метод передаються параметри, від яких залежить, який клас буде створено, то такий *Фабричний Метод* називають *Параметризованим Фабричним Методом*.

В нашому прикладі, класи, що мають бути створені, є *Log4NetLogger* та *EnterpriseLogger*, які імплементують *ILogger*, який широко використовується у нас в аплікації.

Уривок коду 3.1. Інтерфейс *ILogger* та одна із його реалізацій

```
interface ILogger
{
    void LogMessage(string message);
    void LogError(string message);
    void LogVerboseInformation(string message);
}
class Log4NetLogger : ILogger
{
    public void LogMessage(string message)
    {
        Console.WriteLine(string.Format("{0}: {1}", "Log4Net", message));
    }
    // Інші методи не наводимо
```

¹⁵ **Factory Method.** Intent. Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Фабричний метод. Призначення. Визначити інтерфейс для створення об'єкта, але надати підкласам вирішувати який клас інстанціювати. Фабричний метод відделегує інстанціювання своїм підкласам.

Як можна здогадуватися, може статися так, що в майбутньому нам знадобиться додати ще декілька провайдерів логування (мало чого). Як ми уже згадували, вирішення приймається на основі того, що є у файлі конфігурації. Щоб не показувати, який клас ми створюємо, ми делегуємо цю роботу до *LoggerProviderFactory*. І ось як фабрика справляється із цим:

Уривок коду 3.2. Реалізація Фабричного Методу

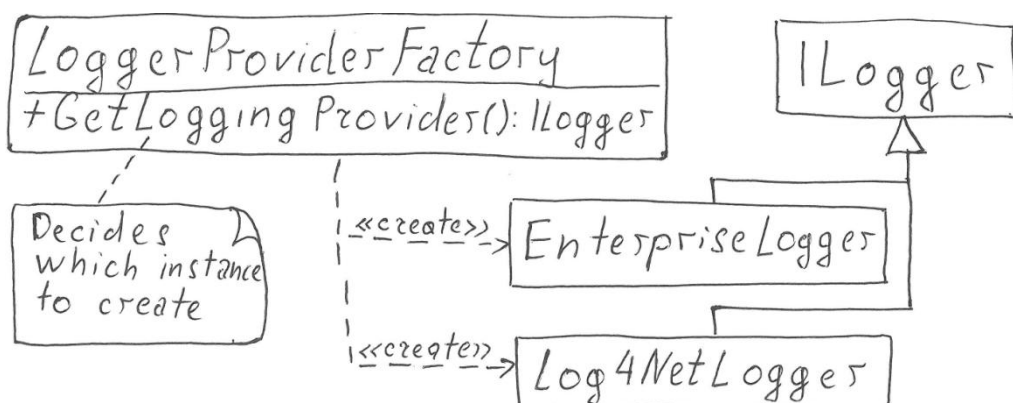
```
class LoggerProviderFactory
{
    public static ILogger GetLoggingProvider(LoggingProviders logProviders)
    {
        switch (logProviders)
        {
            case LoggingProviders.Enterprise:
                return new EnterpriseLogger();
            case LoggingProviders.Log4Net:
                return new Log4NetLogger();
            default:
                return new EnterpriseLogger();
        }
    }
}
```

Те, що ми отримуємо від методу *GetLoggingProvider*, є об'єктом, що реалізує потрібний інтерфейс. *Фабричний Метод* вирішує, який із конкретних класів створювати на основі вхідного параметру. Глянемо на використання:

Уривок коду 3.3. Використання Фабричного Методу

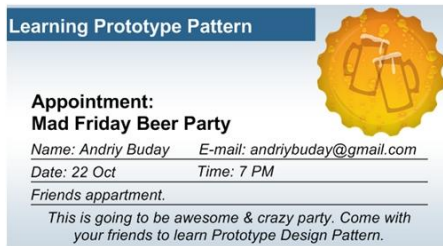
```
public static void Run()
{
    var providerType = GetTypeOfLoggingProviderFromConfigFile();
    ILogger logger = LoggerProviderFactory.GetLoggingProvider(providerType);
    logger.LogMessage("Hello Factory Method Design Pattern.");
    // Вивід: [Log4Net: Hello Factory Method Design Pattern]
}
private static LoggingProviders GetTypeOfLoggingProviderFromConfigFile()
{
    // Це такий собі хадркод, щоб не ускладнювати прикладу
    return LoggingProviders.Log4Net;
}
```

А тепер UML:



UML-діаграма 3. Фабричний Метод

4. Прототип



Чи ви коли небуть працювали із *Outlook*-календарем або ж з якимось іншим, що дозволяє копіювати календарні зустрічі з одного дня на інший?

Для прикладу, уявімо собі, що ваш друг запланував невеличку пивну вечірку на п'ятницю, 22 жовтня, також він виділив час для вечірки із 7-мої вечора до 3-тьої ночі, поставив високий пріоритет, а ще він зазначив, що вечірка в п'ятницю має бути всім до душі. Авжеж, це ж останній робочий день. Оскільки ви були запрошені, вечірка пройшла надзвичайно добре. Під кінець вечірки ваш друг вирішив вислати таке ж запрошення на наступну п'ятницю, але оскільки він уже добряче випив, для нього заповнити календарну форму видалося занадто важко. Яку можливість можна додати в календар, щоб вона була використана вашим другом? Швидше за все «copy-paste» функціональність.

Прототип дозволяє нам створювати копії об'єктів, що уже визначені на стадії дизайну (наприклад, список можливих типів зустрічей) або ж визначаються під час виконання програми («п'ятнична вечірка»), таким чином відпадає необхідність заповняти всі елементи об'єкту від «А до Я». Вже створені або визначені екземпляри об'єкту називаються прототипічними екземплярами (prototypical instances).¹⁶

Ми можемо використовувати цей дизайн патерн для копіювання екземплярів об'єктів в час виконання програми, що дозволяє нам уникати великої кількості похідних класів. Для прикладу, замість того, щоб мати такі класи як «п'ятиповерхова будівля із трикімнатними квартирами», «дев'ятиповерхова будівля із 2-3-4-кімнатними квартирами» і «дванадцятиповерхова будівля із 1-2-3-кімнатними квартирами», ми можемо мати просто 3 екземпляри класу *ApartmentBlock*, ініціалізовані вже з потрібними властивостями, а потім ми просто копіюємо один із екземплярів, коли нашої будівельній компанії потрібно збудувати якийсь конкретний будинок десь в місті. Іншими словами, ми можемо обійтися без написання подібного: «new *ApBlockWith9FloorsAnd234Flats()*» або «new *ApartmentBlock(){Floors = 9; FlatsDic = {2,3,4}}*».

¹⁶ **Prototype**. Intent. Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Прототип. Призначення. Визначає різновиди об'єктів, щоб створити їх на основі прототипічного екземпляру, і створює нові об'єкти копіюючи цей прототип.

Все, чого нам буде достатньо, це `_9FloorsAppBlock.Clone()`. Звичайно, ми можемо поєднювати цей патерн із *Фабричним Методом*, таким чином ми будемо мати метод схожий на `GetMe9FloorsAppBlock()`, який всередині буде викликати копіювання прототипічного екземпляру.

Гляньмо на *Прототун* (*Prototype*), що визначає метод `Clone()` для всіх наших конкретних прототипчиків.

Уривок коду 4.1. Метод `Clone()`

```
class CalendarPrototype
{
    public virtual CalendarPrototype Clone()
    {
        var copyOfPrototype = (CalendarPrototype)this.MemberwiseClone();
        return copyOfPrototype;
    }
}
```

Конкретним прототипом є подія в календарі, яка виглядає приблизно так:

Уривок коду 4.2. Подія в календарі

```
class CalendarEvent : CalendarPrototype
{
    public Attendee[] Attendees { get; set; }
    public Priority Priority { get; set; }
    public DateTime StartDateAndTime { get; set; }

    // Зауважимо, що метод Clone не перевантажений (покищо)
}
```

Клієнтський код (*client code*) виконується тоді, коли ваш друг відкриває календар і правою кнопкою мишки копіює подію, а потім вставляє в інше місце, таким чином автоматично помінявши дату та час початку події.

Уривок коду 4.3. Клієнтський код, що використовує патерн

```
public class PrototypeDemo
{
    public static CalendarEvent GetExistingEvent()
    {
        var beerParty = new CalendarEvent();
        var friends = new Attendee[1];

        var andriy = new Attendee { FirstName = "Andriy", LastName = "Buday" };

        friends[0] = andriy;

        beerParty.Attendees = friends;
        beerParty.StartDateAndTime = new DateTime(2010, 7, 23, 19, 0, 0);
        beerParty.Priority = Priority.High();

        return beerParty;
    }
}
```



```
public static void Run()
{
    var beerParty = GetExistingEvent();
    var nextFridayEvent = (CalendarEvent)beerParty.Clone();
    nextFridayEvent.StartDateAndTime = new DateTime(2010, 7, 30, 19, 0, 0);
    // Про цей код побалакаємо трішки нижче
    nextFridayEvent.Attendees[0].EmailAddress = "andriybuday@liamg.com";
    nextFridayEvent.Priority.SetPriorityValue(0);
    if (beerParty.Attendees != nextFridayEvent.Attendees)
    {
        Console.WriteLine("GOOD: Each event has own list of attendees.");
    }
    if (beerParty.Attendees[0].EmailAddress ==
        nextFridayEvent.Attendees[0].EmailAddress)
    {
        // В цьому випадку добре мати поверхнєву копію кожного із учасників,
        // таким чином моя адреса, ім'я і персональні дані залишаються тими ж
        Console.WriteLine(
            "GOOD: Update to my e-mail address will be reflected in all events.");
    }
    if (beerParty.Priority.IsHigh() != nextFridayEvent.Priority.IsHigh())
    {
        Console.WriteLine(
            "GOOD: Each event should have own priority object, fully-copied.");
    }
}
```

Як бачимо, мій друг зробив копію існуючої події і за допомогою чудового функціоналу *drag-and-drop* змінив дату події. Оскільки я сидів із нашим другом, то помітив, що я можу змінити свою адресу через цю подію, тому я її поміняв на нову, оскільки вона у мене змінилася. Так як багато кому було погано після вечірки, то ми вирішили, що наступного разу не будемо брати аж так багато пива, але оскільки пива не буде дуже багато, то й пріоритет не може бути високим для наступної зустрічі:

Уривок коду 4.4. Зміна пріоритету зустрічі

```
nextFridayEvent.Priority.SetPriorityValue(0);
```

На перший погляд виглядає, що ми отримали те, що хотіли – копію існуючої події із запрошеними, пріоритетом та іншими властивостями. Але коли я відкриваю стару попередню подію, то виявляється, що пріоритет став нейтральним, замість того, щоб бути високим. Як так?

Причина у тому, що, оскільки, ми ще не перевизначали метод *Clone*, для нашого прототипу було виконане поверхнєве копіювання (*MemberwiseClone*) зазначене у базовому класі.

Поверхнєве копіювання (Shallow copy) копіює тільки прямі поля класу, таким чином залишає ті ж посилання, якщо поле було *reference*-типу, а якщо поле було *value*-типу, то буде нова копія.

Глибоке копіювання (Deep copy) копіює ціле дерево об'єктів, таким чином, об'єкти отримують різні фізичні адреси у купі (heap).

Корекція методу Clone у відповідності до наших потреб

Для нашого прототипу ми можемо реалізувати `Clone()` так, як нам заманеться, тому я можу реалізувати *частково глибоке копіювання*. Навіщо це потрібно? Я б не хотів, щоб моя адреса або інші персональні дані були різними для різних подій, але в той час я хочу бути певним, що коли я зробив копію події і змінив пріоритет, то він буде поміняний тільки для цієї події.

В дизайн патерні *Прототип* ми реалізуємо метод `Clone`. Інколи нам може знадобитися повна глибока копія, яку ми можемо досягнути¹⁷ шляхом *ручного копіювання*, що може бути складним, за допомогою *рефлексину*, що може бути повільним, або за допомогою *серіалізації та десеріалізації* в новий екземпляр об'єкту, що також може бути досить дорогим. Але часто вам може знадобитися *частково повне копіювання*, як у нашому прикладі. Ось чому багато мов програмування додали інтерфейс `ICloneable`, що має бути реалізований вами самостійно. Найбільш підходяща реалізація для нашого прикладу є якраз частково повною. Ось як перевантаження клонування може виглядати:

Уривок коду 4.5. Підходящий метод Clone

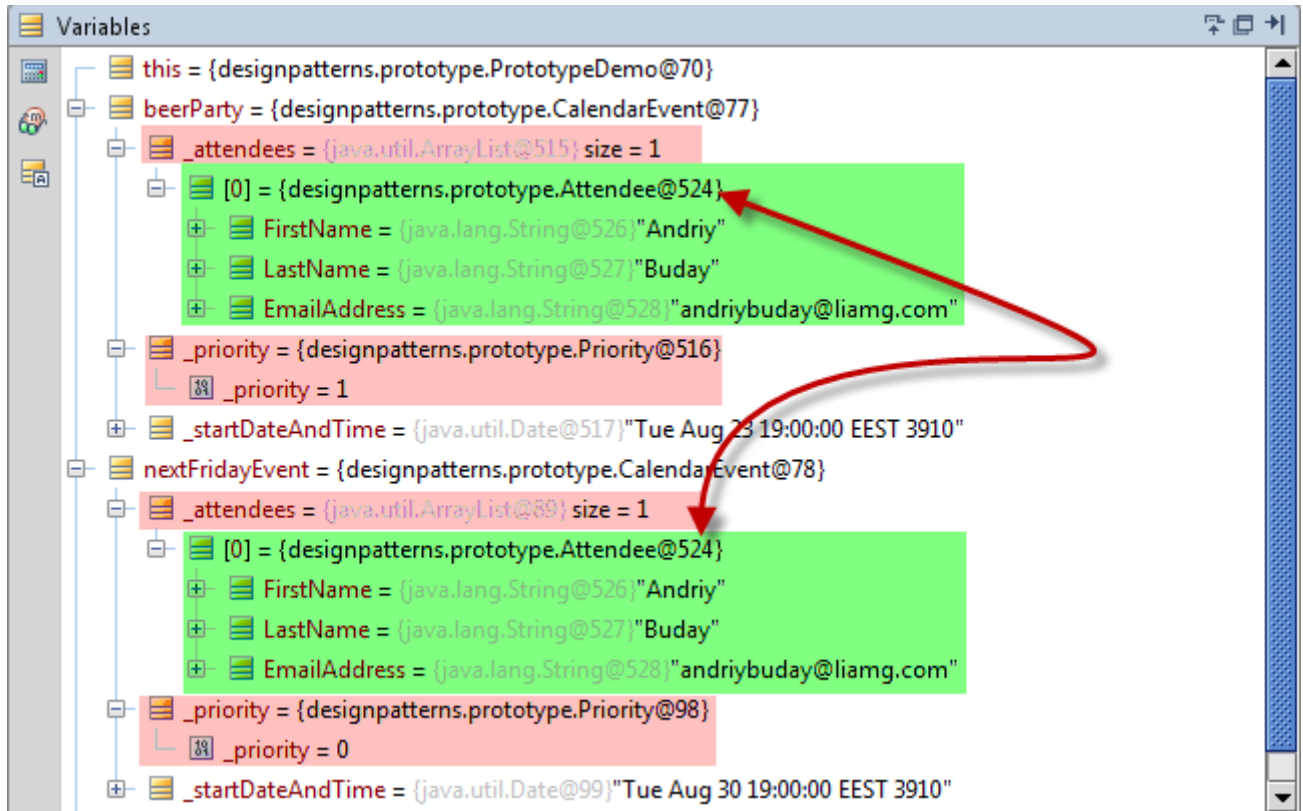
```
public override CalendarPrototype Clone()
{
    var copy = (CalendarEvent)base.Clone();

    // Це дозволить нам мати інший список із посиланнями на тих же відвідувачів
    var copiedAttendees = (Attendee[])Attendees.Clone();
    copy.Attendees = copiedAttendees;
    // Також скопіюємо пріоритет
    copy.Priority = (Priority)Priority.Clone();
    // День і час не варто копіювати – їх заповнять
    // Повертаємо копію події
    return copy;
}
```

На консоль вивелися усі три речення із словом «GOOD» – підтвердження, що все зроблено згідно плану. Оскільки я також маю той же приклад, написаний на *Java*, і в дебаг режимі *IDEA*¹⁸ відображає змінні із однозначними ідентифікаторами посилань (числа після «собачок»), то я ще добавлю скріншот нижче (див. Малюнок 1). Зеленим виділено ті самі елементи із тими ж посиланнями, що і в попередній зустрічі. Червоним підкреслено, що списки людей насправді різні, і на нову подію можуть бути запрошені і інші бажаючі. Також червоним показано спеціально скопійований пріоритет. Дати і події, звичайно, різні.

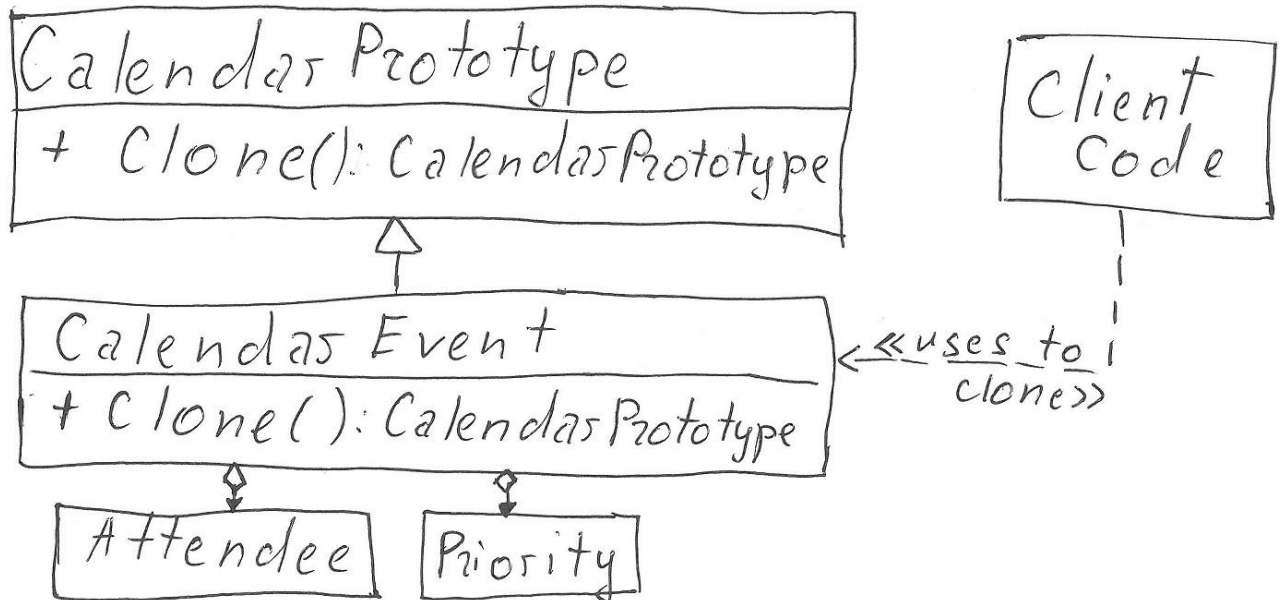
¹⁷ <http://stackoverflow.com/questions/2156120/java-recommended-solution-for-deep-cloning-copying-an-instance>

¹⁸ *IDEA* – середовище розробки для мови програмування *Java*



Малюнок 1. Зображення об'єктів *beerParty* та *nextFridayEvent* в відлагоджувачому режимі IDEA

На цій сторінці є ще трохи місця і для UML-діаграми нашого прикладу.



UML-діаграма 4. Прототип

5. Одинак (Синглтон)



Уявімо, що нам потрібна глобальна логувальна система в програмі, також нам треба логувати наші повідомлення в якийсь один-єдиний файл, причому нумеруючи повідомлення послідовно, незалежно від того, з якого закутка програми вилетіла помилка. Як можна це зробити?

Синглтон забезпечує існування єдиного екземпляру класу та єдиного доступу до нього.¹⁹

Клас, зображений нижче, демонструє просту реалізацію дизайн патерну *Одинак* (ця назва, чесно кажучи, звучить дещо «дико». Думаю, що може буде краще його називати *Синглтон*. Як вважаєте?)

Уривок коду 5.1. Проста реалізація Синглтону

```
class LoggerSingleton
{
    private LoggerSingleton() { }
    private int _logCount = 0;
    private static LoggerSingleton _loggerSingletonInstance =
        new LoggerSingleton();

    public static LoggerSingleton GetInstance()
    {
        return _loggerSingletonInstance;
    }
    public void Log(String message)
    {
        Console.WriteLine(_logCount + ": " + message);
        _logCount++;
    }
}
```

Чіткі ознаки *Одинака* – приватний конструктор та доступ до єдиного, внутрішньо створеного екземпляру, здійснюваний через статичний метод.

Отже, ми збираємося почати роботу програми із методу *DoHardWork* і хочемо залогувати факт того, що вона почалася, всякі інші події та факт того, що робота закінчилася.

Уривок коду 5.2. Наша «важка робота» може виглядати так

```
public static void DoHardWork()
{
    LoggerSingleton logger = LoggerSingleton.GetInstance();
    HardProcessor processor = new HardProcessor(1);
    logger.Log("Hard work started...");
    processor.ProcessTo(5);
    logger.Log("Hard work finished...");
}
```

¹⁹ **Singleton**. Intent. Ensure a class only has one instance, and provide a global point of access to it.

Одинак. Призначення. Забезпечує існування одного екземпляру класу і надає глобальну точку доступу до нього.

Як бачимо метод використовує деякий клас *HardProcessor*. Що саме він насправді робить, нас не дуже переймає, проте ми б хотіли, щоб було залоговано в місцях, коли екземпляр класу створився та коли виконувалися якісь підрахунки.

Уривок коду 5.3. *HardProcessor*

```
class HardProcessor
{
    private int _start;
    public HardProcessor(int start)
    {
        _start = start;
        LoggerSingleton.GetInstance().Log("Processor just created.");
    }
    public int ProcessTo(int end)
    {
        int sum = 0;
        for (int i = _start; i <= end; ++i)
        {
            sum += i;
        }
        LoggerSingleton.GetInstance().Log(
            "Processor just calculated some value: " + sum);
        return sum;
    }
}
```

А ось вивід програми:

```
0: Processor just created.
1: Hard work started...
2: Processor just calculated some value: 15
3: Hard work finished...
```

Я написав цей приклад, коли був у поїзді, і показав його своєму другу, який також є програмістом. Він мені сказав, що я написав *Моностейт*. – «Я? Де?» Ми поглянули на код і виявилось, що тільки одна змінна в *Синглтоні*, яку я використовував, *_logCount*, була статичною в початковому варіанті, який я написав. (Зараз вона змінена на змінну інстансу, тому вгорі дійсно *Синглтон*).

То ж в чому різниця між Синглтоном і Моностейтом?

Синглтон можна розглядати як спосіб забезпечення одного інстансу класу для нашої аплікації. *Моностейт*²⁰ (*Monostate*), взагалі кажучи, робить те ж саме, що і *GoF Синглтон*. В *Моностейті* всі змінні є статичними, отже, теоретично, ви можете мати багато інстансів *Моностейту*, але ж статичні дані одні і ті ж для одного і того ж типу. Таким чином це допомагає також вирішити проблеми багатопоточності.

Давайте глянемо на приклад ще раз. Швидше за все, що логгер не є таким тривіальним, можливо ви захочете додати функціональні зберігання останнього номера логу а при наступному зчитуванні починати нумерацію саме із того

²⁰ <http://c2.com/cgi/wiki?MonostatePattern>

номера. Також ваша програма не така вже й проста і логування викликається не тільки із багатьох місць у коді, але і з багатьох потоків.

Оскільки конструктор вашого класу працює із *IO*, то витрачається час на створення екземпляру класу. Таким чином *GetInstance()* може привезти до ситуації, коли ви створите декілька екземплярів класу (через багатопоточність), а потім, що гірше, будете звертатися до різних логгерів! Напевно, не така вже і гарна ідея для вашої аплікації.

На щастя, є багато способів вирішити цю проблему. Найчастіше використовується *Double-Checked Locking*²¹ (*DCL*). Є багато способів вирішення проблеми і не всі реалізації *DCL* працюють. Прочитайте ось цю дуже хорошу статтю, вказану в зносках²².

Уривок коду 5.4. Подвійне локування для потокобезпечності

```
public class ThreadSafeLoggerSingleton
{
    private ThreadSafeLoggerSingleton()
    {
        // Читаємо дані із якогось файлу і дістаємо номер останнього запису
        // _logCount = вичитане значення
    }
    private int _logCount = 0;
    private static ThreadSafeLoggerSingleton _loggerInstance;
    public static ThreadSafeLoggerSingleton GetInstance()
    {
        if (_loggerInstance == null)
        {
            lock (_loggerInstance)
            {
                if (_loggerInstance == null)
                {
                    _loggerInstance = new ThreadSafeLoggerSingleton();
                }
            }
        }
        return _loggerInstance;
    }
    public void Log(String message)
    {
        Console.WriteLine(_logCount + ": " + message);
        _logCount++;
    }
}
```

Як можна побачити, в методі *Log* я використовую змінну класу і роблю якісь операції, які є майже атомарними. Проте, якщо операції були б складнішими, то синхронізація знадобилася б і там.

²¹ <http://www.javaworld.com/javaworld/jw-05-2001/jw-0525-double.html>

²² <http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>

СТРУКТУРНІ ПАТЕРНИ

Основним завданням *структурних* патернів є формування найбільш підходящої структури та взаємодії між класами для виконання певних завдань.

Якщо потрібно, щоб один об'єкт міг бути зрозумілим під іншим інтерфейсом, використовується *Адаптер*.

Якщо ви хочете розділити абстракцію та імплементацію так, що на одному боці ви матимете абстракцію, а на іншому декілька реалізацій, причому всі доступні до модифікацій, то слід задуматися над поєднанням таких незалежних абстракції та реалізації за допомогою патерну *Міст*.

Якщо елемент містить собі подібні елементи, а вони в свою чергу також можуть містити елементи, то найлегше таку структуру реалізувати за допомогою *Компонувальника*.

Для швидкої та динамічної можливості розширення існуючої функціональності, без зміни її носителя, можна скористатися *Декоратором*.

Якщо ваша система використовує багато об'єктів, що мають спільні дані, то такі дані можна винести та зробити загальнодоступними для економії пам'яті за допомогою патерну *Легковаговик*.

Якщо відсутня можливість працювати із об'єктом напрямку, використайте *Проксі*, що дозволить донести ваші команди до пункту призначення.

6. Адаптер



З деяких причин я дуже довго думав над тим, який же може бути хороший приклад для *Адаптера*. Можна навести приклад, коли є операції, які так і називаються: *ОпераціяА* та *ОпА*, або придумати складний приклад з великою кількістю методів і пропертів в класі, що адаптується... Але весь цей час в думках у мене крутилася така штукенція, яку ми соваємо у розетку, коли вона вузька, ще СРСР-рівська, а ми хочемо запахати шнур для живлення ноутбука із товстою вилкою. Майже, як на ілюстрації, але не настільки жорстока.

Отож, у нас є вилка від зарядного пристрою, яка підходить в широкі роз'єми. В одній із квартир у нас все сучасне, тому *NewElectricitySystem* має метод *MatchWideSocket*, яким ми просто можемо скористатися. В іншій квартирі у нас проблеми, тому *OldElectricitySystem* має тільки метод *MatchThinSocket*. Нажаль ми не можемо собі дозволити взяти дрель і роздвбати отвори в розетці. Натомість ми купуємо адаптер, який надає можливість користуватися тією ж функціональністю споживання електричного струму, але із старої системи.

Адаптер надає можливість користуватися об'єктом, який не є прийнятним у нашій системі і який не можна змінити. Ми адаптуємо його функціональність через інший, відомий нашій системі, інтерфейс.²³

Уривок коду 6.1. Читаємо код і вникаємо

```
// Система яку будемо адаптовувати
class OldElectricitySystem
{
    public string MatchThinSocket()
    {
        return "220V";
    }
}
// Широковикористовуваний інтерфейс нової системи (специфікація до квартири)
interface INewElectricitySystem
{
    string MatchWideSocket();
}
// Ну і власне сама розетка у новій квартирі
class NewElectricitySystem : INewElectricitySystem
{
    public string MatchWideSocket()
    {
        return "220V";
    }
}
```

²³ **Adapter.** Intent. Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Адаптер. Призначення. Конвертувати інтерфейс класу в інший інтерфейс, очікуваний клієнтами. Адаптер дозволяє класам працювати разом, що не могло б бути інакше із-за несумісності інтерфейсів.

```
// Адаптер назовні виглядає як нові євророзетки, шляхом наслідування прийнятого у
// системі інтерфейсу
class Adapter : INewElectricitySystem
{
    // Але всередині він таки знає, що коїлося в СРСР
    private readonly OldElectricitySystem _adaptee;
    public Adapter(OldElectricitySystem adaptee)
    {
        _adaptee = adaptee;
    }
    // А тут відбувається вся магія -
    // наш адаптер «перекладає»
    // функціональність із нового стандарту на старий
    public string MatchWideSocket()
    {
        // Якщо б була різниця в напрузі (не 220V)
        // то тут ми б помістили трансформатор
        return _adaptee.MatchThinSocket();
    }
}

class ElectricityConsumer
{
    // Зарядний пристрій розуміє тільки нову систему
    public static void ChargeNotebook(INewElectricitySystem electricitySystem)
    {
        Console.WriteLine(electricitySystem.MatchWideSocket());
    }
}

public class AdapterDemo
{
    public static void Run()
    {
        // 1)
        // Ми можемо користуватися новою системою без проблем
        var newElectricitySystem = new NewElectricitySystem();
        ElectricityConsumer.ChargeNotebook(newElectricitySystem);

        // 2)
        // Ми повинні адаптуватися до старої системи, використовуючи адаптер
        var oldElectricitySystem = new OldElectricitySystem();
        var adapter = new Adapter(oldElectricitySystem);
        ElectricityConsumer.ChargeNotebook(adapter);
    }
}
```

Попрошу ще раз повернутися до коду і переконатися, що ви прочитали усі коментарі, а взаємодія між класами була зрозуміла. Для кращого засвоєння, візьміть ручку і на будь-якому «огризку» паперу намалюйте UML до цієї реалізації патерну *Адаптер*, а потім узагальніть.

Насправді в реалізації не є обов'язковою композиція *adaptee*. Також наш клас *Adapter* міг би одночасно реалізовувати два інтерфейси – нової і старої системи. А зробивши два конструктори (а чому б і ні?), ми б могли одного разу створити його на базі нової, а іншого – на базі старої системи. Це б дозволило використовувати його в обидва боки. Натиснувши собі якусь кнопку, ваш адаптер перетворюється «шиворіт-на-виворіт».

7. Міст



Уявімо, що ви володієте будівельною компанією, яка будує дачні будинки і житлові масиви. Зазвичай будівлі є двох типів - або з цегли, або з бетонних плит. Оскільки ви бос, то ви вирішили поділити всіх ваших робітників на команди, які будуть вміти робити одні і ті ж операції: *BuildFoundation*, *BuildRoom*, *BuildRoof*. Так як будівлі є двох типів, то вам прийдеється тримати два типи різних бригад.

Одного разу виявилось, що будівлі можуть бути побудовані із бетонних і цегляних стін одночасно. Так як у кожній із бригад вміли працювати тільки із одним типом стін, ви були змушені переміщати цілі бригади із одного закутка міста в інший. Прямо такі переселення народів. Працівники починають жалітися і пропонують вирішення проблеми. Пропозиція полягає у виділенні двох маленьких бригад, які спеціалізуються в будівництві кімнат або із бетонних або із цегляних стін. Таким чином тільки ці бригади і можна буде перевозити із одного закутка міста в інше, замість того, щоб мати окремі великі команди.

Міст дозволяє розділити імплементацію від її абстракції, таким чином реалізація може бути змінена окремо від абстракції, оскільки вона не наслідуються від неї напряму.²⁴

Іншими словами, наш інтерфейс *IBuildingCompany* може мати дві конкретні реалізації, такі як *NearSeeBuildingCompany* та *CityBuildingCompany*, кожна із яких, напевно, якимось по своєму робить фундамент та дах, оскільки ґрунт інший і погода також, але в той же час ми можемо просто і легко змінити реалізацію бригади побудови стін - *WallCreator* для будь-якої із компаній, щоб будувати або цегляні або бетонні стіни.

Уривок коду 7.1. Гляньмо на *BuildingCompany*

```
interface IBuildingCompany
{
    void BuildFoundation();
    void BuildRoom();
    void BuildRoof();
    IWallCreator WallCreator { get; set; }
}

class BuildingCompany : IBuildingCompany
{
    public void BuildFoundation()
    {
        Console.WriteLine("Foundation is built.{0}", Environment.NewLine);
    }
}
```

²⁴ **Bridge**. Intent. Decouple an abstraction from its implementation so that the two can vary independently.

Міст. Призначення. Розділяє абстракцію від її імплементації, таким чином що і те і інше може мінятися незалежно.

```

public void BuildRoom()
{
    WallCreator.BuildWallWithDoor();
    WallCreator.BuildWall();
    WallCreator.BuildWallWithWindow();
    WallCreator.BuildWall();
    Console.WriteLine("Room finished.{0}", Environment.NewLine);
}
public void BuildRoof()
{
    Console.WriteLine("Roof is done.{0}", Environment.NewLine);
}
public IWallCreator WallCreator { get; set; }
}

```

І що тут цікавого? Відповідь – властивість *WallCreator*, яка якраз і є своєрідним мостом до реалізації.

Ще раз замислимося над означенням і прикладом. Методи *BuildFoundation* та *BuildRoof* повністю знаходяться в абстракції і можуть змінюватися, навіть метод *BuildRoom* і той може змінитися, проте він не виконує повноцінну роботу – він надіється на імплементацію, яка прийде по мосту і зробить потрібну роботу потрібним чином.

Уривок коду 7.2. Глянемо на "міст" в дії

```

// Ми маємо дві бригади – одна працює із цеглою, інша із бетоном
var brickWallCreator = new BrickWallCreator();
var concreteSlabWallCreator = new ConcreteSlabWallCreator();

var buildingCompany = new BuildingCompany();
buildingCompany.BuildFoundation();

buildingCompany.WallCreator = concreteSlabWallCreator;
buildingCompany.BuildRoom();

// Компанія може легко переключитися на іншу команду, яка буде будувати
// стіни із інших матеріалів
buildingCompany.WallCreator = brickWallCreator;
buildingCompany.BuildRoom();
buildingCompany.BuildRoom();

buildingCompany.BuildRoof();

```

Хіба це не чудово? Вивід, я припускаю, є інтуїтивним, проте оскільки я не наводив реалізацій класів *BrickWallCreator* та *ConcreteSlabWallCreator* подивимося на нього.

Вивід:

```

Foundation is built.
Concrete slab wall with door.
Concrete slab wall.
Concrete slab wall with window.
Concrete slab wall.
Room finished.

Brick wall with door.
Brick wall.

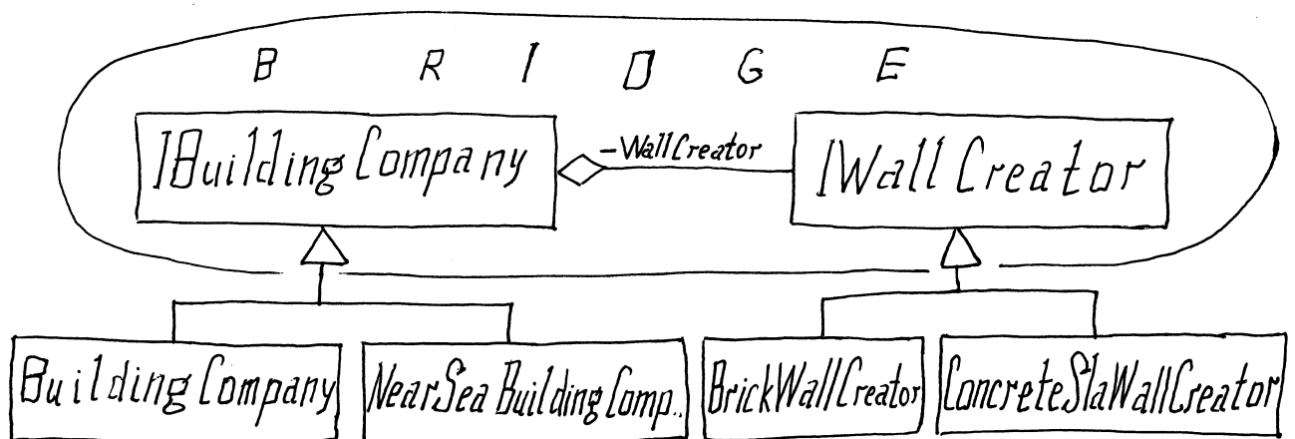
```

Brick wall with window.
Brick wall.
Room finished.

Brick wall with door.
Brick wall.
Brick wall with window.
Brick wall.
Room finished.

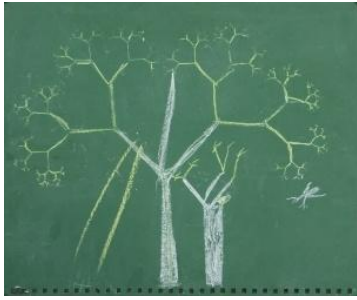
Roof is done.

UML-діаграма зображає структуру цього дизайн патерну. Дивлячись на діаграму, стає зрозумілою його назва.



UML-діаграма 5. Міст

8. Компонувальник



Чи ви коли небузь задумувалися над тим, чому так багато речей у цьому світі мають деревовидну структуру? Адміністративний устрій країни для прикладу, або ж ваша організація. Топ-менеджмент компанії може делегувати роботу менеджерам відділів, які, відповідно, делегують її до ваших прямих менеджерів, а ті, в свою чергу, дадуть вам якусь роботу.

Або для прикладу, XML має деревовидну структуру. Мабуть тому, що це найкращий спосіб зберегти дані, що можуть містити дані, які в свою чергу можуть містити дані, які... Отже, припустимо, що вам слід зкомпонувати якийсь складний документ із частин. Деякі частини вмюють збирати дані (*GatherData*), базуючись на відповідній *ID*-шці. Деякі частини просто утримують інші частини. Ми побудуємо XML «кустарним» способом у цьому прикладі.

Компонувальник дозволяє нам зберігати деревовидну структуру і працювати однаково із батьками та синами у дереві.²⁵

Уривок коду 8.1. інтерфейс що визначає спільні вимоги до батьків і дітей (:

```
interface IDocumentComponent
{
    string GatherData();
    void AddComponent(IDocumentComponent documentComponent);
}
```

А тепер реалізація одного із листків у нашому дереві – *CustomerDocumentComponent*, яка вмє збирати кусок XML, базуючись на *ID*-шці замовника.

Уривок коду 8.2. Реалізація «листокового» компонента

```
class CustomerDocumentComponent : IDocumentComponent
{
    private int CustomerIdToGatherData { get; set; }
    public CustomerDocumentComponent(int customerIdToGatherData)
    {
        CustomerIdToGatherData = customerIdToGatherData;
    }
    public string GatherData()
    {
        string customerData;
        switch (CustomerIdToGatherData)
        {
            case 41:
                customerData = "Andriy Buday";
                break;
        }
    }
}
```

²⁵ **Composite.** Intent. Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Компонувальник. Призначення. Компонує об'єкти в деревовидну структуру для представлення частково-цілої ієрархії. Компонувальник дозволяє розглядати окремі об'єкти і їхні композиції єдиним способом.

```

        default:
            customerData = "Someone else";
            break;
    }
    return string.Format("<Customer>{0}</Customer>", customerData);
}
public void AddComponent(IDocumentComponent documentComponent)
{
    Console.WriteLine("Cannot add to leaf...");
}
}

```

А тепер реалізація нелисткових компонентів нашого дерева. Зверніть увагу, що в методі *GatherData* ми просто ітеруємо по всіх нащадках і викликаємо наш основний метод для збору даних.

Уривок коду 8.3. Компонент, що містить інші компоненти

```

class DocumentComponent : IDocumentComponent
{
    public string Name { get; private set; }
    public List<IDocumentComponent> DocumentComponents { get; private set; }
    public DocumentComponent(string name)
    {
        Name = name;
        DocumentComponents = new List<IDocumentComponent>();
    }
    public string GatherData()
    {
        var stringBuilder = new StringBuilder();
        stringBuilder.AppendLine(string.Format("<{0}>", Name));
        foreach (var documentComponent in DocumentComponents)
        {
            documentComponent.GatherData();
            stringBuilder.AppendLine(documentComponent.GatherData());
        }
        stringBuilder.AppendLine(string.Format("</{0}>", Name));
        return stringBuilder.ToString();
    }
    public void AddComponent(IDocumentComponent documentComponent)
    {
        DocumentComponents.Add(documentComponent);
    }
}

```

Уривок коду 8.3. Клеїмо частинки докупи – процес компонування документу

```

var document = new DocumentComponent("ComposableDocument");
var headerDocumentSection = new HeaderDocumentComponent();
var body = new DocumentComponent("Body");
document.AddComponent(headerDocumentSection);
document.AddComponent(body);

var customerDocumentSection = new CustomerDocumentComponent(41);
var orders = new DocumentComponent("Orders");
var order0 = new OrderDocumentComponent(0);
var order1 = new OrderDocumentComponent(1);
orders.AddComponent(order0);
orders.AddComponent(order1);

body.AddComponent(customerDocumentSection);

```

```
body.AddComponent(orders);  
  
string gatheredData = document.GatherData();  
  
Console.WriteLine(gatheredData);
```

Вивід, певно, схожий на XML. Принаймні я хотів щоб він був таким.

Вивід:

```
<ComposableDocument>  
  <Header>  
    <MessageTime>8:47:23</MessageTime>  
  </Header>  
  <Body>  
    <Customer>Andriy Buday</Customer>  
    <Orders>  
      <Order>Kindle;Book1;Book2</Order>  
      <Order>Phone;Cable;Headset</Order>  
    </Orders>  
  </Body>  
</ComposableDocument>
```

Варто додати, що «GoF-хлопці» пропонують також методи *Remove(Component)* та *GetChild(int)* у інтерфейсі компоненту, тому, можливо, вам захочеться їх додати. Одне хочу сказати - ніколи не зациклюєтися на якихось прикладах і однотипних поясненнях. Ваші потреби можуть дещо відрізнятися від того, що описано в класичному дизайн патерні, але в той же час ваша не класична реалізація вам ідеально буде підходити. Можливо у вас якесь вироджене або ускладнене застосування, зазвичай воно завжди так і є ☺.

9. Декоратор



Розкажу я вам про лікаря, який мав хорошу й швидку машину Mercedes. Оскільки він лікар львівської лікарні, то по дорозі на роботу він часто попадає у корки, тому він спізнюється і це його виводить із себе, в результаті чого страждають... Хто? Його пацієнти. В нашого лікаря є мрія, в якій його машина перетвориться на карету швидкої допомоги і всі інші автомобілі уступатимуть йому дорогу на роботу. Тільки одна проблема із цим: швидка вміє сигналізувати сиреною, а його мерс не має такої можливості. Нам, як програмістам, було б добре *декорувати* (decorate) «мерс», таким чином, щоб він почав голосно «вити». Як ми це можемо зробити?

Декоратор використовується для надання деякої додаткової функціональності нашим об'єктам.²⁶

В нашому прикладі, ми хочемо додати функціональність сирени до конкретної імплементації автомобіля. Нам нічого не заважатиме причепити на машину і газовий розпилювач або що. Отже, маємо базовий клас автомобіля (*Car*):

Уривок коду 9.1. Базовий клас Автомобіля

```
class Car
{
    protected String BrandName { get; set; }
    public virtual void Go()
    {
        Console.WriteLine("I'm " + BrandName + " and I'm on my way...");
    }
}
// Конкретна реалізація класу Car
class Mercedes : Car
{
    public Mercedes()
    {
        BrandName = "Mercedes";
    }
}
```

Для того щоб зберегти контракт базового класу *Car* і мати базовий клас для всіх інших «прибамбасних» функціональностей, створимо *CarDecorator*, що так само наслідується від *Car*. Цей клас і буде основою для патерну *Декоратор*:

²⁶ **Decorator**. Intent. Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Декоратор. Призначення. Приєднує додаткові обов'язки до об'єкта динамічно. Декорування надає гнучку альтернативу наслідуванню в питання розширення функціональності.

Уривок коду 9.2. Декоратор

```
class DecoratorCar : Car
{
    protected Car DecoratedCar { get; set; }
    public DecoratorCar(Car decoratedCar)
    {
        DecoratedCar = decoratedCar;
    }
    public override void Go()
    {
        DecoratedCar.Go();
    }
}
```

Як ми уже зауважили, цей клас агрегує справжній автомобіль, або іншими словами обгортає *DecoratedCar*. Через це даний патерн ще називають *Обгорткою* (або *Wrapper*). Можливо ви десь зустрічали цю ж назву і для дизайн патерну *Адаптер*. Його так само могли називати «*wrapper*». Обидва патерни можуть так називати, але слід розрізняти їхнє призначення. Патерн *Адаптер* застосовується для представлення одного під виглядом іншого, а *Декоратор* для додавання до існуючого більше функціональності, хоч вони обоє агрегують певний об'єкт.²⁷

Настав час для «прибамбасів». Ми додаємо додаткову функціональність до будь-якої машини (чи то «мерс» чи то «беха»), наслідуючися від *CarDecorator* класу. Тут ми додали простий екстеншин біпкання.

Уривок коду 9.3. «Декор» карети швидкої допомоги

```
class AmbulanceCar : DecoratorCar
{
    public AmbulanceCar(Car decoratedCar)
        : base(decoratedCar)
    {
    }
    public override void Go()
    {
        base.Go();
        Console.WriteLine("... beep-beep-beeeeeep ...");
    }
}
```

І власне те, чого ми добивалися

Викорисання виглядає дуже мило - ми покриваємо мерседес «фічами» швидкої і отримуємо мрію лікаря (змінна *doctorsDream*). Опісля ми можемо покрити цей об'єкт ще якимись штуkenціями і, що важливо, ми це все можемо робити динамічно. Сниться лікарю біпкалка – передаємо його мерс у *Декоратор* *AmbulanceCar*, а сниться йому кулемет на даху автомобіля – передаємо у декоратор *ArmorCar*.

²⁷ <http://c2.com/cgi/wiki?DecoratorPattern>

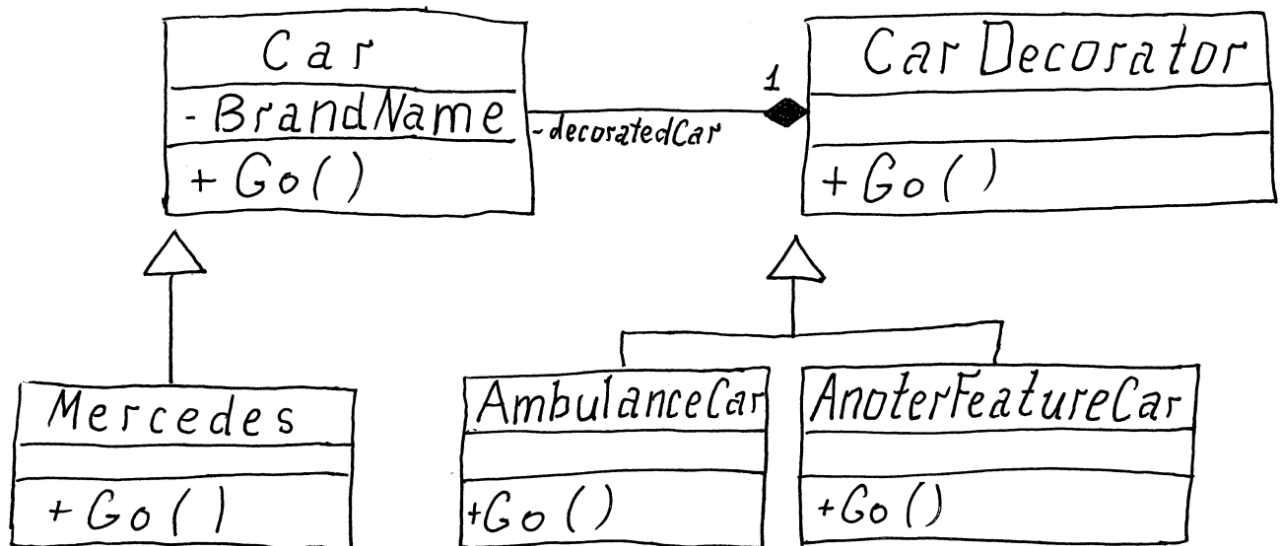
Уривок коду 9.4. Використання патерну

```
var doctorsDream = new AmbulanceCar(new Mercedes());  
doctorsDream.Go();
```

Вивід:

I'm Mercedes and I'm on my way...
... beep-beep-beeeeeer ...

Надіюся що цей вивід був очікуваний. А тепер швидко глянемо на UML цієї мудрості-чудасії:



UML-діаграма 6. Декоратор

Цей патерн має щось спільне і з *Компонувальником* і *Адаптером*. *Адаптер* може змінити інтерфейс поведінки, а *Декоратор* ні (ми наслідуємося від *Car*). *Компонувальник* працює із великою кількістю компонент, а не як *Декоратор* – тільки із однією.

10. Фасад



Уявімо, що ви вирішили провести свої вихідні дуже активно, а тому приїхали на зимовий курорт покататися на лижах. Перед тим як кататися, добре було б пересвідчитися, що ввечері буде де заночувати. Тому ви йдете в місцевий готель і замовляєте собі кімнату, відповідну до ваших вимог. Треба мати на чому кататися, тому ви йдете взяти лижі на прокат. Для того, щоб їх підібрати, у вас запитують вашу вагу, рівень професіоналізму, потім ви підбираєте черевики і палки, для підбору яких у вас ще спитають ваш ріст і розмір взуття. Із цим усім на хребті ви йдете до каси і купуєте абонемент на день. Не знаю як вас, але мене таке ходіння дістало б. Я б хотів якийсь термінал, у який вводили б потрібні дані, оплачуєш і тобі зразу персонал видає черевики, лижі, палки, абонемент на день і ключі до номеру в готелі (і дівчину у номер о_О). Якщо ж мені потрібен лише абонемент на день, а лижне спорядження я маю, то абонемент я купую на місці.

Фасад надає єдину «точку доступу» до підсистеми, тим самим спрощуючи її використання та розуміння.²⁸

В нашому прикладі *Фасадом* буде термінал-обслуговувальна станція (*SkiResortFacade*) а підсистемою є ціла купа прокатних будок, кас і готельних комплексів. Звичайно, ви можете прогулятися по курорту, якщо вам це подобається, але якщо дивитися із точки зору розробки програмного забезпечення, то якщо кожен собі буде «гуляти» куди хоче і як хоче, то до добра це не приведе, а лижники-новачки ніколи не будуть знати куди їм йти в першу чергу.

Уривок коду 10.1. Три системи, доступ до яких надається одним фасадом

```
// 1. Система орендування черевиків
class SkiRent
{
    public int RentBoots(int feetSize, int skierLevel)
    {
        return 20;
    }
    public int RentSki(int weight, int skierLevel)
    {
        return 40;
    }
    public int RentPole(int height)
    {
        return 5;
    }
}
```

²⁸ **Facade.** Intent. Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Фасад. Призначення. Надає єдиний інтерфейс до множини інших інтерфейсів у системі. Фасад визначає верхній інтерфейс, що робить систему легшою до використання.

```
// 2. Система придбання квитків
class SkiResortTicketSystem
{
    public int BuyOneDayTicket()
    {
        return 115;
    }
    public int BuyHalfDayTicket()
    {
        return 60;
    }
}

// 3. Система бронювання місць в готелі
class HotelBookingSystem
{
    public int BookRoom(int roomQuality)
    {
        switch (roomQuality)
        {
            case 3:
                return 250;
            case 4:
                return 500;
            case 5:
                return 900;
            default:
                throw new ArgumentException(
                    "roomQuality should be in range [3;5]");
        }
    }
}

// Фасад, що надає єдиний доступ до всіх систем згаданих вище
class SkiResortFacade
{
    private SkiRent _skiRent = new SkiRent();
    private SkiResortTicketSystem _skiResortTicketSystem
        = new SkiResortTicketSystem();
    private HotelBookingSystem _hotelBookingSystem = new HotelBookingSystem();
    // Беручи до уваги вхідні параметри бронює номер, підбирає лижі і т.д
    // Повертає загальну ціну за все
    public int HaveGoodRest(int height, int weight, int feetSize, int skierLevel,
        int roomQuality)
    {
        int skiPrice = _skiRent.RentSki(weight, skierLevel);
        int skiBootsPrice = _skiRent.RentBoots(feetSize, skierLevel);
        int polePrice = _skiRent.RentPole(height);
        int oneDayTicketPr = _skiResortTicketSystem.BuyOneDayTicket();
        int hotelPrice = _hotelBookingSystem.BookRoom(roomQuality);

        return skiPrice + skiBootsPrice + polePrice + oneDayTicketPr + hotelPrice;
    }
    // Інші методи можуть поєднувати виклики до інших систем
    public int HaveRestWithOwnSkis()
    {
        int oneDayTicketPrice = _skiResortTicketSystem.BuyOneDayTicket();
        return oneDayTicketPrice;
    }
    // Може бути що наш фасад-термінал просто огортає методи із усіх систем
}
```

Цей дизайн патерн можна розглядати як наступний рівень такого важливого принципу як інкапсуляція. Просто на цьому рівні ми інкапсулюємо цілу підсистему. Великі системи зазвичай здійснюють взаємозв'язок одна з одною за

допомогою цього патерну. Космічна станція у космосі стикається з іншою за допомогою одного механізму, а не прикрученням сотні дротів поокремо. Також добрим тоном буде, якщо кожна із збірок, які ви пишете, має свого роду *Фасад* із відкритих класів та інтерфейсів, щоб цю збірку потім можна було легко використовувати із інших частин програми.

11. Легковаговик (Флайвейт)



Уявіть, що ви розробляєте якогось бота до онлайн-іграшки. У вас уже є веб-клієнт, який на кожну відповідь від сервера парсить HTML, у якому є записані юніти гри. Гра має близько 50 різних типів юнітів-тваринок, але коли ви розбираєте відповідь, то ви можете отримати цілу гору екземплярів однієї і тієї ж тваринки і ще цілу купу екземплярів інших тварюк.

Якщо ваш бот шпіляє дуже інтенсивно, то аплікація буде просто заганятися кожного разу створювати велику кількість інстансів кожного із юнітів. Але що цікаво, самі юніти мають одні і ті ж початкові значення *здоров'я* та одне й те ж саме *зображення*.

Звичайно, із плином гри *здоров'я* зменшується, але *зображення*, що візуалізує тварюку, одне й те ж саме!

Це може призвести до неефективного використання пам'яті. То як ми можемо зробити загальну інформацію про зображення тваринки (і т.п.) спільною для кожного окремого юніта одного типу?

Флайвейт забезпечує підтримку великої кількості об'єктів шляхом виокремлення спільної інформації для збереження в одному екземплярі.²⁹

Цього разу приклад буде побудованний по іншому – ми розглянемо декілька варіантів вирішення проблеми.

Перший варіант із створенням об'єктів зображення для кожного юніта

Уривок коду 11.1. Базовий клас для юнітів

```
abstract class Unit
{
    public string Name { get; protected set; }
    public int Health { get; protected set; }
    public Image Picture { get; protected set; }
}
```

І два породжені класи, які зображають гобліна (*Goblin*) та дракона (*Dragon*) з їхніми початковими значеннями *здоров'я* (*Health*) та *зображення* (*Picture*). Наведемо код гобліна. Тут слід звернути увагу на те, що зображення гобліна є дуже велике, тому буде займати багато пам'яті.

²⁹ **Flyweight.** Intent. Use sharing to support large numbers of fine-grained objects efficiently.

Флайвейт. Призначення. Ефективна підтримка великої кількості об'єктів шляхом виділення спільної інформації.

Уривок коду 11.2. Код гобліна ☺

```
class Goblin : Unit
{
    public Goblin()
    {
        Name = "Goblin";
        Health = 8;
        Picture = UnitImagesFactory.CrateGoblinImage();
    }
}
```

Уривок коду 11.3. Парсер насправді імітує якусь роботу по створенню об'єктів

```
class Parser
{
    public List<Unit> ParseHTML()
    {
        var units = new List<Unit>();
        for (int i = 0; i < 150; i++)
            units.Add(new Dragon());
        for (int i = 0; i < 500; i++)
            units.Add(new Goblin());
        Console.WriteLine("Dragons and Goblins are parsed from HTML page.");
        return units;
    }
}
```

То ж ми створили 150 Драконів і 500 Гоблінів. Це забрало нам аж... 439 Mb.

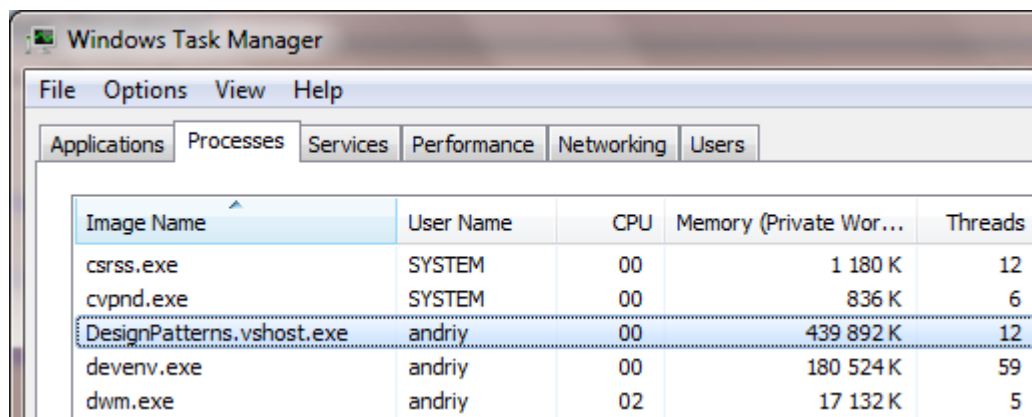


Image Name	User Name	CPU	Memory (Private Wor...	Threads
csrss.exe	SYSTEM	00	1 180 K	12
cvpnd.exe	SYSTEM	00	836 K	6
DesignPatterns.vshost.exe	andriy	00	439 892 K	12
devenv.exe	andriy	00	180 524 K	59
dwm.exe	andriy	02	17 132 K	5

Малюнок 2. Пам'ять зайнята юнітами гри без застосування патерну Флайвейт

Другий варіант. То як же Флайвейт працює? (не думаю, що когось пів гіга жертвої пам'яті влаштує)

Просто введемо новий клас, який буде фабрикою зображень. У нашому випадку зображення і буде *флайвейт об'єктом*. Але варто зауважити, що насправді замість зображення ми б могли «шарити» більше інформації, скажімо, ми б мали базовий клас *UnitInitialInfo*, який був би полем у класі *Unit*, а потім фабрика видавали б нам конкретні реалізації цього *Info* класу. В нашому прикладі ми наводимо тільки створення «імейджу» для різних тваринок, причому якщо зображення уже завантажувалося для істоти, то ми його знову не будемо завантажувати.

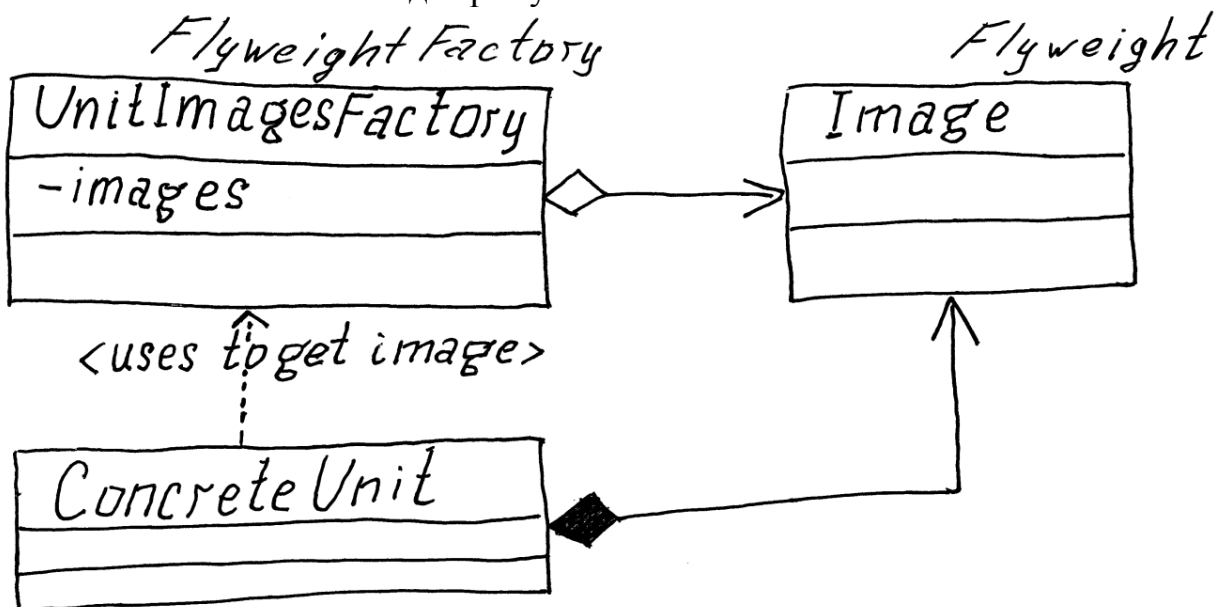
Увок коду 11.4. Сховище для зображень

```
class UnitImagesFactory
{
    public static Dictionary<Type, Image> Images = new Dictionary<Type, Image>();
    public static Image CrateDragonImage()
    {
        if (!Images.ContainsKey(typeof(Dragon)))
        {
            Images.Add(typeof(Dragon), Image.Load("Dragon.jpg"));
        }
        return Images[typeof(Dragon)];
    }
    public static Image CrateGoblinImage()
    {
        if (!Images.ContainsKey(typeof(Goblin)))
        {
            Images.Add(typeof(Goblin), Image.Load("Goblin.jpg"));
        }
        return Images[typeof(Goblin)];
    }
}
```

Увок коду 11.5. Конструктори Гобліна і Дракона дещо змінені для використання нашої фабрики

```
class Dragon : Unit
{
    public Dragon()
    {
        Name = "Dragon";
        Health = 50;
        // От власне те, що змінилося від попередньої версії
        Picture = UnitImagesFactory.CrateDragonImage();
    }
}
```

Також глянемо на UML діаграму.



UML-діаграма 7. Патерн Флайвейт

Ця UML діаграма не відповідає класичній діаграмі із GoF книжки, але треба сказати, що нам слід бути до цього готовими. В реальному світі реалізація

патерну часто відрізняється від того, що описано у всіма відомій книзі. Можна дуже дивуватися тому, що люди чітко пробують дотриматися такої ж структури - часто вона буває занадто загальною. Як на мене, то в оригінальній статті про *Flyweight* не було очевидним, що *Goblin* та *Dragon* є репрезентаціями *Info*-класів для тваринок.

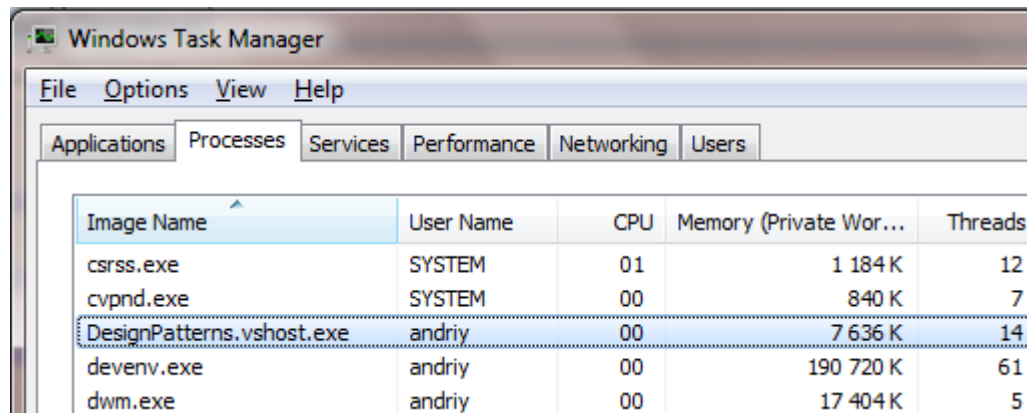


Image Name	User Name	CPU	Memory (Private Wor...	Threads
csrss.exe	SYSTEM	01	1 184 K	12
cvpnd.exe	SYSTEM	00	840 K	7
DesignPatterns.vshost.exe	andriy	00	7 636 K	14
devenv.exe	andriy	00	190 720 K	61
dwm.exe	andriy	00	17 404 K	5

Малюнок 3. Тепер в ран-таймі наш чудо-бот зжерас тільки 7 Мб

Як видно із малюнка вище, ми домоглися скорочення споживання пам'яті нашою програмою-ботом.

12. Проксі



Спробуйте себе уявити у процесі знешкодження бомби? Невже не лячно? Навіть якщо вам важко таке уявити або ви вважаєте, що й так ніколи нічого подібного не будете робити, комусь все ж таки приходится знешкоджувати бомби час від часу. Можу поручитися, що сапери, за винятком тих які хильнули чвертку, також

відчують страх перед вибухівкою. На наше щастя зараз є багато різних технологій, які дозволяються обійтися без присутності людини під час знешкодження вибухового матеріалу. Більше того, робот може проробити більш складні операції аніж людина, оскільки він точніший і потужніший (проте не настільки повороткий). Принаймні в робота не трясуться руки на відміну від сапера напідпитку.

Комунікація із таким роботом звісно безпроводна, керування здійснюється за допомогою кінекту³⁰, або якогось костюму, або, принаймні, супер джойстиків. Робот також має на собі панель управління, яка дозволяє керувати ним безпосередньо на місці, на випадок, якщо якийсь внучок Бен Ладена буде блокувати сигнал до робота.

Проксі підміняє реальний об'єкт та надсилає запити до нього тоді, коли це потрібно. Проксі також може ініціалізувати реальний об'єкт, якщо він до того не існував.³¹

Говорячи про приклад вище, реальним об'єктом є робот (*RobotBombDefuser*). Це купа важкого залізаччя, яким ми можемо керувати за допомогою віддалених контролів (*RobotBombDefuserProxy*). Так як чудо-кінект установка передає сигнали роботу, вона, звісно, знає як приєднатися до справжнього робота і передавати йому сигнали від ваших рухів (щось подібне відбувається коли ви приєднуєтеся до віддалених сервісів і викликаєте методи на них).

Проксі також часто використовується коли потрібна *лінива ініціалізація*³². В такому випадку реальний об'єкт не буде створений, допоки не буде звертання до його методів. Всередині проксі методу добавляється перевірка, чи справжній об'єкт ще не ініціалізований, і якщо ні, то тоді відбувається ініціалізація.

³⁰ Зовсім недавно побачив відео, у якому була зображена ідея використання кінекту для керування робота-знешкоджувача. Був здивований. <http://www.xbox.com/en-US/Kinect/Kinect-Effect>

³¹ **Proxy**. Intent. Provide a surrogate or placeholder for another object to control access to it.
Проксі. Призначення. Надає замітника або утримувача для іншого об'єкту, щоб керувати ним.

³² http://uk.wikipedia.org/wiki/Lazy_initialization

Давайте глянемо на наш приклад. Ось сам робот.

Уривок коду 12.1. Робот-знешкоджувач бомб

```
class RobotBombDefuser
{
    private Random _random = new Random();
    private int _robotConfiguredWavelength = 41;
    private bool _isConnected = false;

    public void ConnectWireless(int communicationWaveLength)
    {
        if(communicationWaveLength == _robotConfiguredWavelength)
        {
            _isConnected = IsConnectedImmitatingConnectivityIssues();
        }
    }
    public bool IsConnected()
    {
        _isConnected = IsConnectedImmitatingConnectivityIssues();
        return _isConnected;
    }
    private bool IsConnectedImmitatingConnectivityIssues()
    {
        // Імітуємо погане з'єднання (працює в 4 із 10 спробах)
        return _random.Next(0, 10) < 4;
    }
    public virtual void WalkStraightForward(int steps)
    {
        Console.WriteLine("Did {0} steps forward...", steps);
    }
    public virtual void TurnRight()
    {
        Console.WriteLine("Turned right...");
    }
    public virtual void TurnLeft()
    {
        Console.WriteLine("Turned left...");
    }
    public virtual void DefuseBomb()
    {
        Console.WriteLine("Cut red or green or blue wire...");
    }
}
```

Основними методами, що роблять усю роботу є *WalkStraightForward*, *TurnRight*, *TurnLeft*, *DefuseBomb*. Є також методи, які здійснюють безпроводне з'єднання та виконують перевірку на його наявність (*IsConnected*). Вони потрібні для більшої реалістичності цього прикладу. Можливо, вони несуть додатковий «шум», якщо так, то можна їх упустити при читанні коду.

А ось реалізація самого *Проксі*. *Проксі* завжди має посилання на реальний об'єкт і часто наслідується від того ж класу, яким є реальний об'єкт.

Уривок коду 12.2. Проксі до робота – можливість ним керувати із сторони

```
class RobotBombDefuserProxy : RobotBombDefuser
{
    private RobotBombDefuser _robotBombDefuser;
    private int _communicationWaveLength;
    private int _connectionAttempts = 3;

    public RobotBombDefuserProxy(int communicationWaveLength)
    {
        _robotBombDefuser = new RobotBombDefuser();
        _communicationWaveLength = communicationWaveLength;
    }
    public virtual void WalkStraightForward(int steps)
    {
        EnsureConnectedWithRobot();
        _robotBombDefuser.WalkStraightForward(steps);
    }
    public virtual void TurnRight()
    {
        EnsureConnectedWithRobot();
        _robotBombDefuser.TurnRight();
    }
    public virtual void TurnLeft()
    {
        EnsureConnectedWithRobot();
        _robotBombDefuser.TurnLeft();
    }
    public virtual void DefuseBomb()
    {
        EnsureConnectedWithRobot();
        _robotBombDefuser.DefuseBomb();
    }
    private void EnsureConnectedWithRobot()
    {
        if (_robotBombDefuser == null)
        {
            _robotBombDefuser = new RobotBombDefuser();
            _robotBombDefuser.ConnectWireless(_communicationWaveLength);
        }
        for (int i = 0; i < _connectionAttempts; i++)
        {
            if (_robotBombDefuser.IsConnected() != true)
            {
                _robotBombDefuser.ConnectWireless(_communicationWaveLength);
            }
            else
            {
                break;
            }
        }
        if(_robotBombDefuser.IsConnected() != true)
        {
            throw new BadConnectionException("No connection with remote bomb
diffuser robot could be made after few attempts.");
        }
    }
}
```

Таким чином, наш проксі просто передає запити до справжнього об'єкту і завжди перед цим перевіряє чи з'єднання не пропало, і якщо пропало, то пробує тричі його відновити. Якщо і після цього не виходить, то генерується помилка.

Ось використання проксі:

Уривок коду 12.3. Операція знешкодження (у випадку невдачі – план «В»)

```
public static void Run()
{
    int opNum = 0;
    try
    {
        var proxy = new RobotBombDefuserProxy(41);
        proxy.WalkStraightForward(100);
        opNum++;
        proxy.TurnRight();
        opNum++;
        proxy.WalkStraightForward(5);
        opNum++;
        proxy.DefuseBomb();
        opNum++;

        Console.WriteLine();
    }
    catch (BadConnectionException e)
    {
        Console.WriteLine("Exception has been caught with message: ({0}).  
Decided to have human operate robot there.", e.Message);
        PlanB(opNum);
    }
}

private static void PlanB(int nextOperationNum)
{
    RobotBombDefuser humanOperatingRobotDirectly = new RobotBombDefuser();

    if(nextOperationNum == 0)
    {
        humanOperatingRobotDirectly.WalkStraightForward(100);
        nextOperationNum++;
    }
    if (nextOperationNum == 1)
    {
        humanOperatingRobotDirectly.TurnRight();
        nextOperationNum++;
    }
    if (nextOperationNum == 2)
    {
        humanOperatingRobotDirectly.WalkStraightForward(5);
        nextOperationNum++;
    }
    if (nextOperationNum == 3)
    {
        humanOperatingRobotDirectly.DefuseBomb();
    }
}
```

В кодї вище наведено використання проксі для робота, а також "план Б", якщо неможливо приєднатися до робота. В такому випадку створється пряме

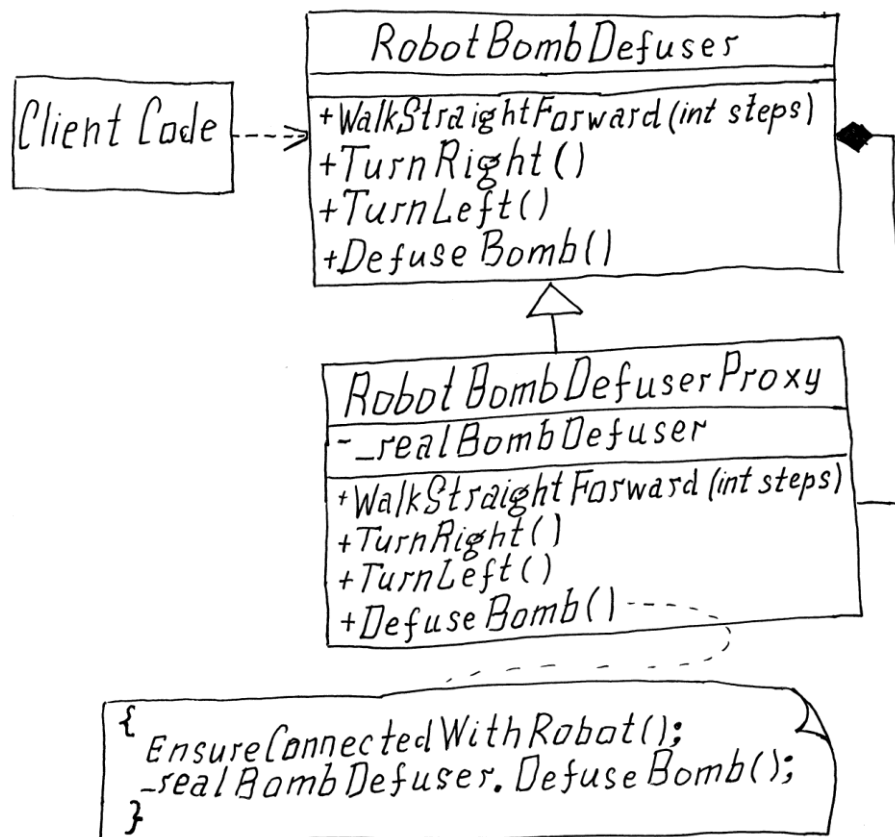
посилання на робота і методи виконуться уже безпосередньо на ньому. Код може виглядати трохи складо через *opNum* та *nextOperationNum*, які були додані, щоб імітувати довиконання операції після втрати зв'язку.

Вивід:

```
Did 100 steps forward...
Turned right...
Exception has been caught with message: (No connection with remote bomb diffuser
robot could be made after few attempts.). Decided to have human operate robot
there.
Did 5 steps forward...
Cut red or green or blue wire...
```

Моя імплементація відрізняється від пропонованої у книзі «банди чотирьох» тим, що *Проксі* і справжній об'єкт не наслідуються від одного суперкласу або інтерфейсу. Але я вважаю, що така реалізація тепер частіше зустрічається. Особливо у різних фреймворках. Наприклад, якийсь «мокінг»³³ фреймворк вимагають від вас мати віртуальні методи. Я думаю, що тепер ви здогадуєтеся чому.

А ось UML-діаграма чудасії, яку ми закодили:



UML-діаграма 8. Проксі

³³ <http://martinfowler.com/articles/mocksArentStubs.html>

ПАТЕРНИ ПОВЕДІНКИ

Ще однією групою патернів є такі, що акцентують свою увагу на поведінці. Вони або інкапсулюють поведінку, або дозволяють її розподілити.

Щоб забезпечити почергову передачу роботи від одного класу до іншого і так далі, аж до поки робота не буде виконана, використовують **Ланцюжок Відповідальностей**.

Інколи краще запакувати інформацію про дії, які слід виконати, в один об'єкт **Команди** і переслати на опрацювання, або ж просто виконати в потрібному місці.

Багато явищ можна описати за допомогою якоїсь спеціальної мови, наприклад погодні умови можуть бути записані значками, зрозумілими тільки метеорологам, але якщо вам подана граматики цієї мови і пояснення значків, цілком можливо, що ви зможете **Інтерпретувати** метеорологічне речення і зрозуміти його суть.

Колекції об'єктів можуть бути «хитрими» і містити багато підколекцій та поокремих об'єктів. Щоб спростити життя користувачу такої колекції та щоб не викривати логіки колекції, придумали **Ітератор**, який допомагає легко і грамотно обійти усі об'єкти всередині.

Спрощення координації роботи між деякою кількістю об'єктів може бути досягнуте виділенням посередника або медіатора. **Медіатором** може бути ваш бригадир на будівництві або ваш менеджер.

Можливість повернутися до попереднього стану системи має велике значення. Така функціональність може бути досягнута **Хранителем**.

Зверху завжди видніше, що коїться внизу. **Спостерігач** допоможе централізувати огляд роботи декількох класів та генерувати відповідні події.

Стан системи та умови переходу між ними можуть бути винесені в окремі класи для легшого контролю над цією системою. Все це досягається за допомогою дизайн патерну **Стан**.

Піти на право чи на ліво, а чи взагалі кудись йти? Відповідь на це питання може залежати від певних параметрів і є нічим іншим як певною **Стратегією**.

Втомились від одноманітної роботи, яка завжди шаблонна, окрім деталей, які час від часу міняються? Віддайте цю роботу **Шаблонному Методу**.

Коли потрібно виконати деякі дії над об'єктом, причому вони кожного разу різні, такі дії можуть бути винесені в окремі класи-відвідувачі. Опісля ваш об'єкт може приймати **Відвідувачів** для виконання конкретних дій.

13. Ланцюжок Відповідальностей



Уявіть, що ви пішли із своїми друзями в кафе. Кафе дещо специфічне – мало місця, і коли вам приносять якусь страву, зазвичай доводиться передавати її наступній людині за столом. Ваш найкращий друг сів найближче до краю, тому першим він і отримує до рук замовлення. Так як він мало спав зранку і любить поїсти м'ясця, то він ніколи не передесть вам м'ясної страви чи кави, допоки сам не насититься. Наступним після друга сидите ви, а далі ваша подружка, яка знаходиться біля стіни. Вона отримає все останньою, але, на щастя, вона хоче тільки капучіно, та й передавати їй уже не треба нікому.

Ланцюжок Відповідальностей забезпечує обробку об'єкта, шляхом передачі його по ланцюжку доти, доки не буде здійснена обробка якоюсь із ланок.³⁴

Я думаю що весь механізм патерну *Ланцюжка Відповідальностей* є зрозумілим – ми маємо набір обробників (*handlers*) або відвідувачів кафе, які вміють обробляти команду (*command*) – їжу у нашому випадку. Якщо обробити команду не вдається, то вона передається наступному обробітнику.

Для прикладу із нашим кафе, загальним інтерфейсом відвідувача такого дивного кафе може бути такий базовий клас:

Уривок коду 13.1. Дивний відвідувач кафе (обробник)

```
abstract class WierdCafeVisitor
{
    public WierdCafeVisitor CafeVisitor { get; private set; }
    protected WierdCafeVisitor(WierdCafeVisitor cafeVisitor)
    {
        CafeVisitor = cafeVisitor;
    }
    public virtual void HandleFood(Food food)
    {
        // Якщо не в змозі подужати їжу, передаємо її ближчому другові
        if (CafeVisitor != null)
        {
            CafeVisitor.HandleFood(food);
        }
    }
}
```

³⁴ **Chain of Responsibility.** Intent. Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Ланцюжок Відповідальностей. Призначення. Уникає зв'язності відправника запиту із його адресатом, шляхом надання іншим об'єктам можливість обробити запит. Передає отримані об'єкти вздовж ланцюжка допоки якась ланка не обробить об'єкт.

Як бачимо, по замовчуванню їжа просто передається до наступного відвідувача у ланцюжку, якщо такий є.

Глянемо на реалізацію, яка найбільше підходить вашому вибагливому другові:

Уривок коду 13.2. Конкретний дивний відвідувач кафе – ваш кращий друг

```
class BestFriend : WierdCafeVisitor
{
    public List<Food> CoffeeContainingFood { get; private set; }
    public BestFriend(WierdCafeVisitor cafeVisitor) : base(cafeVisitor)
    {
        CoffeeContainingFood = new List<Food>();
    }

    public override void HandleFood(Food food)
    {
        if(food.Ingradients.Contains("Meat"))
        {
            Console.WriteLine(
                "BestFriend: I just ate {0}. It was tasty.",
                food.Name);
            return;
        }
        if (food.Ingradients.Contains("Coffee") && CoffeeContainingFood.Count < 1)
        {
            CoffeeContainingFood.Add(food);
            Console.WriteLine(
                "BestFriend: I have to take something with coffee. {0} looks fine.",
                food.Name);
            return;
        }
        base.HandleFood(food);
    }
}
```

Реалізації ще двох обробітників – *Me* та *GirlFriend* мають бути зрозумілими, але все ж таки наведемо реалізацію відвідувача-подружки:

Уривок коду 13.3. Ваша подружка

```
class GirlFriend : WierdCafeVisitor
{
    public GirlFriend(WierdCafeVisitor cafeVisitor) : base(cafeVisitor)
    {
    }

    public override void HandleFood(Food food)
    {
        if(food.Name == "Cappuccino")
        {
            Console.WriteLine("GirlFriend: My lovely cappuccino!!!");
            return;
        }
        // Базовий виклик base.HandleFood(food) для останнього обробітника-дівчини
        // не має сенсу, тому можна викинути експешин або нічого не робити
    }
}
```

Все відносно просто – дівчина хоче капучіно, але вона у ланцюжку остання, тому допоки ваш друг, який перший у ланцюжку, не вип’є щось із кофейном, капучіно вона не отримає.

А тепер глянемо на все у дії. Створимо два капучіно, два супи (один м’ясний) і кусок м’яса, створимо наших відвідувачів кафе та будемо подавати їжу в руки друзів:

Уривок коду 13.3. Замовляємо і передаємо

```
var cappuccino1 = new Food("Cappuccino", new List<string> {"Coffee", "Milk",  
                                                         "Sugar"});  
var cappuccino2 = new Food("Cappuccino", new List<string> {"Coffee", "Milk"});  
var soup1 = new Food("Soup with meat", new List<string> {"Meat", "Water",  
                                                         "Potato"});  
var soup2 = new Food("Soup with potato", new List<string> {"Water", "Potato"});  
var meat = new Food("Meat", new List<string> {"Meat"});  
  
var girlFriend = new GirlFriend(null);  
var me = new Me(girlFriend);  
var bestFriend = new BestFriend(me);  
  
bestFriend.HandleFood(cappuccino1);  
bestFriend.HandleFood(cappuccino2);  
bestFriend.HandleFood(soup1);  
bestFriend.HandleFood(soup2);  
bestFriend.HandleFood(meat);
```

Вивід:

```
BestFriend: I have to take something with coffee. Cappuccino looks fine.  
GirlFriend: My lovely cappuccino!!!  
BestFriend: I just ate Soup with meat. It was testy.  
Me: I like Soup. It went well.  
BestFriend: I just ate Meat. It was testy.
```

Як видно із виводу в консоль, дівчина отримала тільки друге капучіно, а ви були змушені їсти суп без м’яса :)

Що цікаво, мо ми можемо після моєї дівчини підчепити ще один обробітник – скажімо мішечок для собачки. Туди скидатимемо те, що ніхто не захоче їсти. Для цього прийдеться трішки змінити клас, щоб він мав метод на подібі *SetNextCafeVisitor* для динамічної додачі обробників.

14. Команда



Ваш бос дуже вимогливий – він ніколи не переймається тим, як буде робитися робота і не особливо переймається тим, хто її буде робити – йому головне щоб вона була зроблена, як тільки замовник дасть добро. Проте вашому босу ніхто не заважає назначати людей, які будуть працювати над виконанням завдання. Він вирішив, що ви, оскільки ви на високих позиціях у компанії, ідеально підходите для того, щоб зібрати бригаду, отримати список вимог від замовника і бути готовими запустити роботу, як тільки замовник підпише контракт.

Команда дозволяє інкапсулювати всю інформацію, необхідну для виконання певних операцій, які можуть бути виконані пізніше, використавши об'єкт команди.³⁵

В нашому прикладі ви є *Командою* (*command*) – тому що ви інкапсулюєте справжню робочу групу і параметри, необхідні для старту роботи (проект і вимоги). Бригада, що є отримувачем (*receiver*) роботи, а також вимоги і інші параметри, були вам передані вашим босом (*клієнтським кодом*).

Замовник має зв'язок із вами і може попросити вас виконати всю необхідну роботу як тільки запуститься процес після підписання контракту. Замовник – це ваш запускатч (*invoker*), він знає як попросити вас виконати роботу тоді, коли це йому зручно.

А зараз трошки глянемо на код. Ось клієнтський код, що презентує вашого боса:

Уривок коду 14.1. Використання патерну – клієнтський код

```
// Замовник
var customer = new Customer();
// Із певних міркування, бос завжди знає, що грошей стає тільки
// на бригаду Z
var team = new Team("Z");
// Також бос отримав список вимог, що треба буде передати бригаді
var requirements = new List<Requirement>() { new Requirement("Cool web site"),
                                             new Requirement("Ability to book beer on site") };
// Ви повинні бути готові бути викликаними замовником
ICommand commandX = new YouAsProjectManagerCommand(team, requirements);
// Передача вас у «найми» замовнику 😊
customer.AddCommand(commandX);
```

³⁵ **Command.** Intent. Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Команда. Призначення. Інкапсулює запит в об'єкт, таким чином даючи можливість параметризувати клієнтський код різними запитами, залогувати, або закинути в чергу чи підтримувати необроблювані операції.

```
// В компанії також є програміст-герой, що кодує на швидкості світла
var heroDeveloper = new HeroDeveloper();
// Бос вирішив віддати йому проект A
ICommand commandA = new HeroDeveloperCommand(heroDeveloper, "A");
customer.AddCommand(commandA);

// Як тільки замовник підписує контракт із вашим босом,
// ваша бригада і програміст-герой готові виконати все, що треба
// згідно вихідного коду контракту
customer.SignContractWithBoss();
```

Нижче наведені два приклади конкретної реалізації команд.

Уривок коду 14.2. Команда

```
public interface ICommand
{
    // Кожна Команда має метод для її запуску
    void Execute();
}
// Приклад однієї із Команд до виконання
class YouAsProjectManagerCommand : ICommand
{
    public YouAsProjectManagerCommand(Team team, List<Requirement> requirements)
    {
        Team = team;
        Requirements = requirements;
    }
    public void Execute()
    {
        // Реалізація делегує роботу до потрібного отримувача
        Team.CompleteProject(Requirements);
    }
    protected Team Team { get; set; }
    protected List<Requirement> Requirements { get; set; }
}
// І ще один приклад
class HeroDeveloperCommand : ICommand
{
    public HeroDeveloperCommand(HeroDeveloper heroDeveloper, string projectName)
    {
        HeroDeveloper = heroDeveloper;
        ProjectName = projectName;
    }
    public void Execute()
    {
        // Реалізація делегує роботу до потрібного отримувача
        HeroDeveloper.DoAllHardWork(ProjectName);
    }
    protected HeroDeveloper HeroDeveloper { get; set; }
    public string ProjectName { get; set; }
}
```

Team та *HeroDeveloper* є отримувачами роботи, яка має бути зроблена. Об'єкти цих класів передаються в команди разом із іншими параметрами. Як на диво, програмісту-герою достатньо назви проекту для його виконання, а вам (*YouAsProjectManagerCommand*) все ж таки потрібні вимоги.

Замовник насправді не дуже переймається тим, хто буде робити його роботу, але він чітко знає що хоче її зробити, тому він викликає метод *Execute* для кожної доступної йому команди, зразу ж після підписання контракту.

Уривок коду 14.3. Замовник – володіє командами і запускає їх

```
class Customer
{
    protected List<ICommand> Commands { get; set; }
    public Customer()
    {
        Commands = new List<ICommand>();
    }
    public void AddCommand(ICommand command)
    {
        Commands.Add(command);
    }
    public void SignContractWithBoss()
    {
        foreach (var command in Commands)
        {
            command.Execute();
        }
    }
}
```

Вивід:

```
User Story (Cool web site) has been completed
User Story (Ability to book beer on site) has been completed
Hero developer completed project (A) without requirements in manner of
couple hours!
```

Взаємодія між цими всіма класами є точно така ж, як і в класичному поясненні патерну, тому діаграми не наводжу – її легко знайти в інтернеті. Для асоціацій використовуйте англomовні назви в дужках.

Але із тієї самої причини, що це класичний приклад, взаємодія є досить громіздка. Тому пам'ятайте ключові моменти – команда інкапсулює інформацію і надає назовні один метод для виконання дій. Сама команда, як об'єкт, може бути переслана до будь-яких закутків вашого коду, де її можуть успішно використати.

15. Інтерпретер



Якась уявна компанія веде дуже дивний бізнес. Вони скуповують всяке барахло, яке люди виставляють під час так званих «гаражних» розпродаж, набивають ними вантажівки, а в кінці робочого дня пробують порахувати на скільки вони того добра набрали і за скільки зможуть продати в інший час у іншому місці. Оскільки більшість речей із розпродажу запакована у якісь старі упаковки, а також тому, що багато чого має хаотичний спосіб розташування у вантажівці, інколи важко порахувати загальну ціну. Але якщо ми знаємо поточний контекст цін, граф речей у вантажівці, то це зробити можна.

Компанія також унікальна тим, що відповідальність за рахунок загальної суми покладається на вантажівку. Певним чином вона знає, що повинна додати ціну усіх упаковок, кожна упаковка знає, що вона повинна додати ціну чи то упаковок чи то речей і т.д.

Інтерпретер дозволяє описати граматику певної мови, за допомогою чого можна записати речення на цій мові та інтерпретувати його значення.³⁶

Говорячи про граматику нашого дещо «висмоктаного з пальця» прикладу, мовою буде вантажівка/упаковка/різні речі, реченням буде поточне заповнення вантажівки, а значенням речення буде загальна ціна речей всередині.

Існує два види виразів у мові: такі які можна зрозуміти одразу, такі вирази називаються термінальними (*terminal expressions*), та такі, які вимагають застосування граматичних правил мови. Останні називають нетермінальними виразами (*nonterminal expressions*).

Проводячи паралелі із нашим прикладом, термінальним виразом буде старий телевізор, ноут або ліжко, оскільки ми зараз знаємо ціну на них. А нетермінальним виразом буде упаковка, у яку закинули три ноутбуки. В такому випадку необхідно буде порахувати скільки коштуватиме вміст такої упаковки.

Глянемо на вихідний код прикладу:

Уривок коду 15.1. Ось вирази які використовуються у нашій мові

```
// Абстрактний вираз
abstract class Goods
{
    public abstract int Interpret(CurrentPricesContext context);
}
```

³⁶ **Interpreter.** Intent. Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Інтерпретер. Призначення. Маючи мову, визначає представлення її граматики та інтерпретатор, що використовує це представлення, щоб інтерпретувати речення цієї мови.

```
// Нетермінальний вираз (необхідна логіка для визначення значення)
class GoodsPackage : Goods
{
    public List<Goods> GoodsInside { get; set; }
    public override int Interpret(CurrentPricesContext context)
    {
        var totalSum = 0;
        foreach (var goods in GoodsInside)
        {
            totalSum += goods.Interpret(context);
        }
        return totalSum;
    }
}
// Термінальний вираз (зразу повертає значення взявши із його із контексту)
class TV : Goods
{
    public override int Interpret(CurrentPricesContext context)
    {
        int price = context.GetPrice("TV");
        Console.WriteLine("TV: {0}", price);
        return price;
    }
}
// Інші термінальні вирази (Laptop, Bed)
```

Як можна побачити *GoodsPackage* знає про те, як себе подати, а саме, коли він просумує ціну речей всередині. У нашому прикладі мова дуже проста, тільки з одним правилом, але у інших мовах усе може бути набагато складніше. Для прикладу, у якійсь уявній мові, пов'язаній із обрахунками, нетермінальними виразами зможуть бути звичайні “+”, “-”, “/”, “*”, “Sqrt”, “Integral”, або ще щось інше. Така мова також може мати ширший вибір термінальних виразів.

Залишилися ще дві речі, які відіграють помітну роль у патерні. Цей контекст (*context*) зберігає глобальну інформацію для процесу інтерпретування. У нашому прикладі контекст сьогоднішніх цін у певному місті дозволить порахувати сукупну ціну товарів. Ще однією роллю у патерні є клієнт, який відповідає за прочитання речення та виклик методу інтерпретації. Нижче наводиться тільки код клієнту, оскільки код контексту не є дуже важливим.

Уривок коду 15.1. Інтерпретер у дії

```
class InterpreterDemo
{
    public static void Run()
    {
        new InterpreterDemo().RunInterpreterDemo();
    }

    public void RunInterpreterDemo()
    {
        // Дістаємо синтаксичне дерево, що представляє речення
        var truckWithGoods = PrepareTruckWithGoods();
        // Отримуємо останній контекст цін
        var pricesContext = GetRecentPricesContext();
        // Інтерпретуємо
        var totalPriceForGoods = truckWithGoods.Interpret(pricesContext);
    }
}
```

```
        Console.WriteLine("Total: {0}", totalPriceForGoods);
    }

    private CurrentPricesContext GetRecentPricesContext()
    {
        var pricesContext = new CurrentPricesContext();
        pricesContext.SetPrice("Bed", 400);
        pricesContext.SetPrice("TV", 100);
        pricesContext.SetPrice("Laptop", 500);
        return pricesContext;
    }

    public GoodsPackage PrepareTruckWithGoods()
    {
        var truck = new GoodsPackage() { GoodsInside = new List<Goods>() };

        var bed = new Bed();
        var doubleTriplePackedBed = new GoodsPackage()
        {
            GoodsInside = new List<Goods>() { new GoodsPackage() {
                GoodsInside = new List<Goods>() { bed } } }
        };
        truck.GoodsInside.Add(doubleTriplePackedBed);
        truck.GoodsInside.Add(new TV());
        truck.GoodsInside.Add(new TV());
        truck.GoodsInside.Add(new GoodsPackage()
        {
            GoodsInside = new List<Goods>() {
                new Laptop(), new Laptop(), new Laptop()
            }
        });

        return truck;
    }
}
```

А ось вивід:

```
Bed: 400
TV: 100
TV: 100
Laptop: 500
Laptop: 500
Laptop: 500
Total: 2100
```

І ще одне досить важливе: *Інтерпретер* – це такий дизайн патерн, який, швидше за все, вам ніколи не пригодиться у житті завдяки своєму дещо специфічному застосуванню.

16. Ітератор



Уявіть, що ви розробник стратегічної воєнної гри. Армія має складну структуру: вона складається із героя і трьох груп. Коли генерал видає указ і ресурси щоб полікувати всіх воїнів (герой також є воїном), ви хочете проітерувати по всіх солдатах і викликати метод *Treat()* на кожному екземплярі.

Як це можна зробити легко і без вникання в структуру армії?

Ітератор дозволяє доступатися по чергово до елементів будь-якої колекції без вникання в суть її імплементації.³⁷

Таким чином в застосуванні до нашої проблеми, ми хочемо щоб *SoldiersIterator* «пробігся» по всіх солдатах. Код нижче показує використання *Ітератора*.

Уривок коду 16.1. Використання патерну – ітерування по армії

```
var iterator = new SoldiersIterator(earthArmy);
while (iterator.HasNext())
{
    var currSoldier = iterator.Next();
    currSoldier.Treat();
}
```

Як бачимо, ми отримали екземпляр класу ітератора *SoldiersIterator*. І простим циклом проходимося по всіх солдатах армії. Це дуже легко, що і є основним завданням ітератора.

Армія складається із одного героя і може містити багато груп, кожна із яких може містити багато солдатів. Отже, як ми бачимо, структура армії складна і деревовидна. Код нижче показує створення армії:

Уривок коду 16.2. Структура армії

```
var andriybuday = new Hero("Andriy Buday");
var earthArmy = new Army(andriybuday);

var groupA = new Group();
for (int i = 1; i < 4; ++i) groupA.AddNewSoldier(new Soldier("Alpha:" + i));
var groupB = new Group();
for (int i = 1; i < 3; ++i) groupB.AddNewSoldier(new Soldier("Beta:" + i));
var groupC = new Group();
for (int i = 1; i < 2; ++i) groupC.AddNewSoldier(new Soldier("Gamma:" + i));

earthArmy.AddArmyGroup(groupB);
earthArmy.AddArmyGroup(groupA);
earthArmy.AddArmyGroup(groupC);
```

³⁷ **Iterator.** Intent. Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Ітератор. Призначення. Надає послідовний доступ до елементів об'єкта-агрегата без висвітлення його внутрішньої структури.

Герой (*Hero*) – це клас унаслідуваний від солдата (*Soldier*) і основна різниця полягає в тому, що він має вищий початковий рівень здоров'я.

Уривок коду 16.3. Солдат та солдат-герой

```
class Soldier
{
    public String Name;
    public int Health;
    private const int SoldierHealthPoints = 100;
    protected virtual int MaxHealthPoints { get { return SoldierHealthPoints; } }

    public Soldier(String name)
    {
        Name = name;
    }
    public void Treat()
    {
        Health = MaxHealthPoints;
        Console.WriteLine(Name);
    }
}
class Hero : Soldier
{
    private const int HeroHealthPoints = 500;
    protected override int MaxHealthPoints { get { return HeroHealthPoints; } }

    public Hero(String name)
        : base(name)
    {
    }
}
```

То ж, якщо ми можемо рухатися по складній колекції так легко, де ж вся складність? Звісно, вона інкапсульована в конкретному класі ітератора.

Уривок коду 16.4. Ітератор і складна логіка ітерування (яку не раджу читати)

```
class SoldiersIterator
{
    private readonly Army _army;
    private bool _heroIsIterated;
    private int _currentGroup;
    private int _currentGroupSoldier;

    public SoldiersIterator(Army army)
    {
        _army = army;
        _heroIsIterated = false;
        _currentGroup = 0;
        _currentGroupSoldier = 0;
    }

    public bool HasNext()
    {
        if (!_heroIsIterated) return true;
        if (_currentGroup < _army.ArmeyGroups.Count) return true;
        if (_currentGroup == _army.ArmeyGroups.Count - 1)
            if (_currentGroupSoldier < _army.ArmeyGroups[_currentGroup].Soldiers.Count)
                return true;

        return false;
    }
}
```

```

public Soldier Next()
{
    Soldier nextSoldier;
    if (_currentGroup < _army.ArmyGroups.Count)
    {
        // В кожній групі ітеруємо по кожному солдату
        if (_currentGroupSoldier < _army.ArmyGroups[_currentGroup].Soldiers.Count)
        {
            nextSoldier =
                _army.ArmyGroups[_currentGroup].Soldiers[_currentGroupSoldier];
            _currentGroupSoldier++;
        }
        else
        {
            _currentGroup++;
            _currentGroupSoldier = 0;
            return Next();
        }
    }
    // Герой останнім покидає поле бою
    else if (!_heroIsIterated)
    {
        _heroIsIterated = true;
        return _army.ArmyHero;
    }
    else
    {
        // Викидуємо виняток
        throw new Exception("End of colletion");
    }
    return nextSoldier;
}
}

```

Цей приклад дещо відхиляється від стандартного. Я собі поставив за мету підкреслити головне завдання, яке вирішує паттерн. Головною різницею між моїм поясненням і тоннами інших пояснень є те, що стандартні є більш абстраговані. Я створював потрібний нам ітератор таким чином:

Уривок коду 16.5. Просте створення ітератора

```
var iterator = new SoldiersIterator(earthArmy);
```

Але зазвичай створення ітератора також інкапсулюється під методом агрегата. Це потрібно в тих ситуаціях, коли ми хочемо мати декілька різних ітераторів (скажімо, ще один тільки для героїв). Мій код міг би виглядати так:

Уривок коду 16.6. Класичне створення ітератора

```
AbstractIterator iterator = AbstractArmy.GetSoldiersIterator();
```

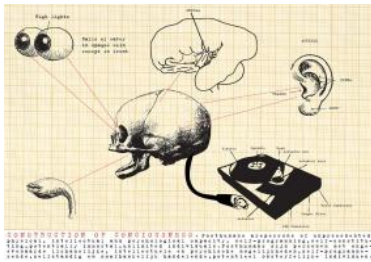
Уривок коду 16.7. В .NET є готові інтерфейси *IEnumerable* та *IEnumerator*, що нам допомагають

```

var list = new List<int>();
// GetEnumerator це метод інтерфейсу IEnumerable (агрегат)
var enumerator = list.GetEnumerator();
// MoveNext метод інтерфейсу IEnumerator і буде методом ітератора
enumerator.MoveNext();

```


17. Медіатор



Я трохи думав над тим, який має бути приклад для *Медіатора* і мені на думку нічого кращого не спадало, ніж класичний приклад із взаємодією елементів в користувацькому інтерфейсі. Потім ще декілька варіантів прокрутилися, аж поки не спала на думку взаємодія нейронів. Я ще трохи подумав і в голову прийшла геніальна ідея (звичайно я не ручаюся, що більше такого прикладу немає, але мене осінило ним). Наш мозок є медіатором до різних частин тіла. Мозок ідеально підходить під опис дизайн патерну *Медіатор*.

Просто спробуйте уявити, якби кожна із частин вашого тіла знала одна про іншу. Якщо б ваше око бачило щось приємне, то воно мало б знати, як напругу зв'язатися із ногами і змусити їх рухатися у заданому напрямку. Або якщо б вас хтось вдарив у живіт, ваш живіт повинен був би навчитися захищатися руками. Живіт може й боліти, тоді він буде змушений знати про цілу систему м'язів, щоб змусити тіло прийняти розслаблююче положення. Взаємодія, описана вище, як багато-до-багатьох не є природньою для нашого тіла. Проте, чомусь, вона часто застосовна деякими програмістами до їхнього коду. Спочатку, поки програміст все пише по свіжому, такий код працює нормально, але із часом він перетворється на суцільне спагетті³⁸ – безлад, в якому розібратися важко, а змінити поведінку, не поломавши чогось, також складно.

Наше тіло має одну центральну систему, яка аналізує прийняті сигнали і здійснює потрібні реакції. Це можна застосувати і до коду, який ми пишемо.

Медіатор централізує взаємодію між компонентами, таким чином послаблюючи їхню зв'язність.³⁹

Медіатор елегантно спрощує розуміння взаємодії між компонентами. Це полегшує підтримку коду у майбутньому, але, оскільки логіка централізована, вона може стати досить складною, зрештою чим наш мозок і є.

Нижче, наперед, наведемо консольний вивід, демонструючий приклад в дії:

```
Enter body part ('Ear', 'Eye', 'Hand' or empty to exit):  
Ear  
Enter what you hear:  
You are cool!  
FACE: Smiling...
```

³⁸ http://en.wikipedia.org/wiki/Spaghetti_code

³⁹ **Mediator**. Intent. Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Медіатор. Призначення. Визначає об'єкт, що інкапсулює взаємодію між множиною об'єктів. Медіатор покращує слабкозв'язність шляхом утримання об'єктів від прямих посилок один на одного, а також дозволяє вам незалежно змінювати взаємодію.

```

Enter body part ('Ear', 'Eye', 'Hand' or empty to exit):
Hand
What you feel is soft? (Yes/No)
Yeah!
What you feel is hurting? (Yes/No)
No
LEG: Stepping forward...
HAND: Embracing what is in front of you...
Enter body part ('Ear', 'Eye', 'Hand' or empty to exit):
Ear
Enter what you hear:
You are dumbass stupid guy!
LEG: Stepping forward...
HAND: Just hit offender...
LEG: Just kicked offender in front of you...
Enter body part ('Ear', 'Eye', 'Hand' or empty to exit):
Hand
What you feel is soft? (Yes/No)
No
What you feel is hurting? (Yes/No)
Yes
LEG: Stepping back...

```

Як видно, мозок знає як діяти в різних ситуаціях і які частини тіла слід задіяти. То як це відбувається?

Мозок (або новоспечений *Медітор*) знає про кожну частину тіла (*colleague*), а кожна частина тіла знає про мозок, тому може передавати сигнали йому та самій приймати їх. Звичайно, кожна частина тіла виконує ще й свою безпосередню функцію.

Уривок коду 17.1. Базовий клас для частин тіла *colleague* (знає про мозок)

```

class BodyPart
{
    private readonly Brain _brain;
    public BodyPart(Brain brain)
    {
        _brain = brain;
    }
    public void Changed()
    {
        _brain.SomethingHappenedToBodyPart(this);
    }
}

```

Уривок коду 17.2. Конкретна реалізація *colleague* може виглядати так

```

class Ear : BodyPart
{
    private string _sounds = string.Empty;
    public Ear(Brain brain) : base(brain) { }

    public void HearSomething()
    {
        Console.WriteLine("Enter what you hear:");
        _sounds = Console.ReadLine();

        Changed();
    }
}

```

```

    public string GetSounds()
    {
        return _sounds;
    }
}

```

Як бачимо, вухо (*Ear*) може чути (*HearSomething*) і може передати звуки на аналіз мозку (*GetSounds*). Деякі частини тіла мають іншу функціональність. Як, для прикладу, реалізація класу обличчя (*Face*):

Уривок коду 17.3. Обличчя

```

class Face : BodyPart
{
    public Face(Brain brain)
        : base(brain)
    {
    }
    public void Smile()
    {
        Console.WriteLine("FACE: Smiling...");
    }
}

```

Як і слід було очікувати, клас медіатора є досить громіздким, оскільки він відповідає за «розрулювання» ситуації. Можливо, вам не сподобається те, як реалізовано цей конкретний мозок, але це не є аж на стільки важливо. Важливо зрозуміти як він діє.

Уривок коду 17.4. Мозок, або медіатор

```

// Медіатор
class Brain
{
    public Brain()
    {
        CreateBodyParts();
    }

    private void CreateBodyParts()
    {
        Ear = new Ear(this);
        Eye = new Eye(this);
        Face = new Face(this);
        Hand = new Hand(this);
        Leg = new Leg(this);
    }

    public Ear Ear { get; private set; }
    public Eye Eye { get; private set; }
    public Face Face { get; private set; }
    public Hand Hand { get; private set; }
    public Leg Leg { get; private set; }

    public void SomethingHappenedToBodyPart(BodyPart bodyPart)
    {
        if (bodyPart is Ear)
        {
            string heardSounds = ((Ear)bodyPart).GetSounds();

```

```

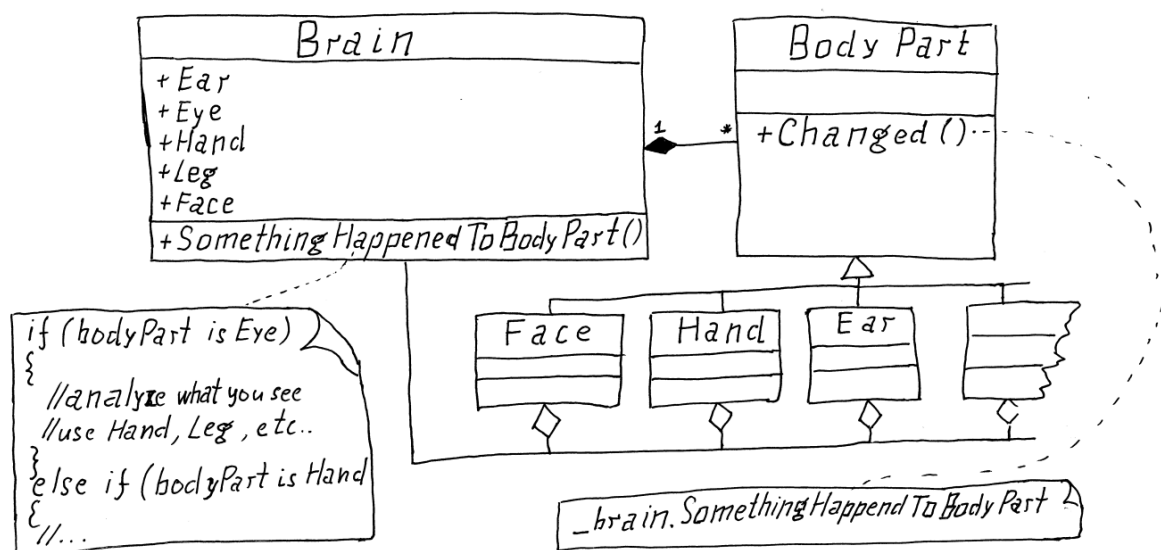
if (heardSounds.Contains("stupid"))
{
    // Атакуємо образника
    Leg.StepForward();
    Hand.HitPersonNearYou();
    Leg.Kick();
}
else if (heardSounds.Contains("cool"))
{
    Face.Smile();
}
}
else if (bodyPart is Eye)
{
    // Мозок може проаналізувати, що ви бачите і
    // прореагувати відповідно, використовуючи різні частини тіла
}
else if (bodyPart is Hand)
{
    var hand = (Hand)bodyPart;

    bool hurtingFeeling = hand.DoesItHurt();
    if (hurtingFeeling)
    {
        Leg.StepBack();
    }

    bool itIsNice = hand.IsItNice();
    if (itIsNice)
    {
        Leg.StepForward();
        Hand.Embrace();
    }
}
else if (bodyPart is Leg)
{
    // Якщо на ногу впаде цегла, змінюємо вираз обличчя ☺
}
}
}

```

Додаю трохи UML:



UML-діаграма 9. Медіатор

18. Хранитель (Мemento)



Якщо ви коли небузь бавилися в «стрілялки», то дуже вірогідно, що ви знайомі із значенням хот-кїїв F5 та F9. І, навіть, якщо ви таки не мали шансу в житті погратися в «шпільки», ідея швидкого збереження поточного стану і відновлення до нього ідеологічно є знайомою (навіть якщо це було Ctrl+Z у програмі Word). Натискаючи F5 ви зберігаєте поточне місце знаходження і рівні життя/броні та, можливо, ще якусь інформацію, наприклад, скільки монстрів було вже вбито на даній позиції (напевно для того, щоб не «мочити» їх заново). Коли клавіша F9 натискається відбувається повернення до попереднього збереженого стану.

Під час збереження стану, швидше за все, вам не дуже хочеться, щоб ця інформація була доступна іншими класами (інкапсуляція стану), таким чином ви будете певні, що ніхто не зменшить рівень життя. Також може знадобитися функціональність по збереженню послідовності станів. Наприклад, можливість повернутися на 2 або 3 збереження назад натискаючи Shift+F9 (+F9). Як це можна реалізувати?

Хранитель використовується тоді, коли ви хочете відмінити операції без відображення внутрішньої структури Хазяїна (Originator - гра у нашому прикладі). Координація операцій здійснюється Опікуном (Caretaker - контроллер гри), який надає можливість простого збереження миттєвих станів системи без уявлення що ці стани собою являють.⁴⁰

Давайте глянемо на реалізацію:

Уривок коду 18.1. Гра

```
public class GameOriginator
{
    // Стан містить здоров'я та к-ть вбитих монстрів
    private GameState _state = new GameState(100, 0);

    public void Play()
    {
        // Імітуємо процес гри -
        // здоров'я повільно погіршується, а монстрів стає все менше
        Console.WriteLine(_state.ToString());
        _state = new GameState((int)(_state.Health*0.9), _state.KilledMonsters + 2);
    }
}
```

⁴⁰ **Memento.** Intent. Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Хранитель. Призначення. Без порушення інкапсуляції зафіксувати та відокремити внутрішній стан об'єкта так, що об'єкт потім зможе повернутися до цього стану.

```
public GameMemento GameSave()
{
    return new GameMemento(_state);
}

public void LoadGame(GameMemento memento)
{
    _state = memento.GetState();
}
}
```

Уривок коду 18.2. Хранитель GameMemento

```
public class GameMemento
{
    private readonly GameState _state;

    public GameMemento(GameState state)
    {
        _state = state;
    }

    public GameState GetState()
    {
        return _state;
    }
}
```

Таким чином, цей клас може згенерувати екземпляр *Хранителя* із поточним знімком (станом) гри, в той же час можна витягнути стан із уже існуючого *Хранителя*, але при цьому ніхто більше не буде працювати із станом гри напряму.

Нижче наведений код може тільки завантажити останній збережений «сейв», але все можна легко вдосконалити.

Уривок коду 18.3. Відповідальний

```
public class Caretaker
{
    private readonly GameOriginator _game = new GameOriginator();
    private readonly Stack< GameMemento > _quickSaves = new Stack< GameMemento >();

    public void ShootThatDumbAss()
    {
        _game.Play();
    }

    public void F5()
    {
        _quickSaves.Push(_game.GameSave());
    }

    public void F9()
    {
        _game.LoadGame(_quickSaves.Peek());
    }
}
```

Припустимо що гра протікала так, як у наступному уривку коду:

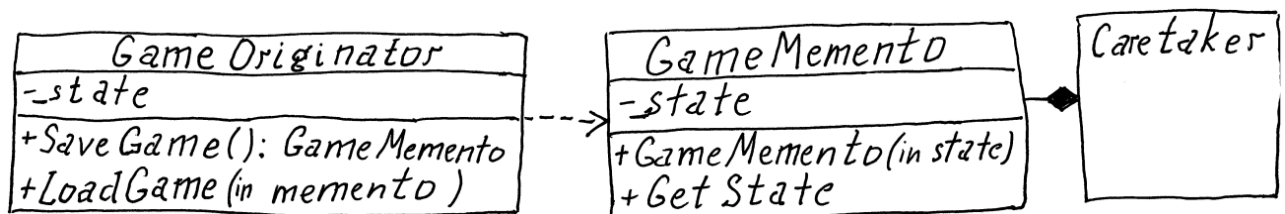
Уривок коду 18.4. Використання

```
var caretaker = new Caretaker();  
caretaker.F5();  
caretaker.ShootThatDumbAss();  
caretaker.F5();  
caretaker.ShootThatDumbAss();  
caretaker.ShootThatDumbAss();  
caretaker.ShootThatDumbAss();  
caretaker.ShootThatDumbAss();  
caretaker.F9();  
caretaker.ShootThatDumbAss();
```

В такому випадку наша демонстраційна програма згенерує такий вивід (*State* має перевантажений метод *ToString*):

```
Health: 100  
Killed Monsters: 0  
Health: 90  
Killed Monsters: 2  
Health: 81  
Killed Monsters: 4  
Health: 72  
Killed Monsters: 6  
Health: 64  
Killed Monsters: 8  
Health: 90  
Killed Monsters: 2
```

А ось проста UML діаграма, намальована для цього прикладу:



UML-діаграма 10. Хранитель стану гри

19. Спостерігач



Багато людей люблять дивитися бокс. Але окрім цього хтось те все діло фінансує. Левова частка фінансів приходить від реклами та трансляційних дозволів, а також від всяких фанів і азартних гравців, які програють солідні суми, роблячи ставки. Уявімо боксерський бій і двох людей, що роблять ставки – один любить ризиковані

ставки, а інший, навпаки, дуже консервативний, і завжди ставить на того, хто, швидше за все, виграє бій. Хто такі ці гравці?

Не важко здогадатися, що гравці – це спостерігачі (*observer*), а бійка, на яку вони ставлять гроші, є суб'єктом споглядання (*subject*). Вони постійно спостерігають за ходом бійки, щоб змінити свої ставки.

Спостерігач дозволяє автоматично реагувати багатьом об'єктам на зміну стану певного іншого об'єкта.⁴¹

Отже, кожен поважаючий себе азартний гравець може оновити свої ставки коли буде потрібно, тому він має метод *Update()*.

Уривок коду 19.1. Спостерігач (Observer) із реалізаціями (RiskPlayer та ConservativePlayer)

```
interface IObservable
{
    void Update(IObservable subject);
}
class RiskPlayer : IObservable
{
    public string BoxerToPutMoneyOn { get; set; }
    public void Update(IObservable subject)
    {
        var boxFight = (BoxFight)subject;
        if (boxFight.BoxerAScore > boxFight.BoxerBScore)
            BoxerToPutMoneyOn = "I put on boxer B, if he win I get more!";
        else BoxerToPutMoneyOn = "I put on boxer A, if he win I get more!";

        Console.WriteLine("RISKYPLAYER:{0}", BoxerToPutMoneyOn);
    }
}
class ConservativePlayer : IObservable
{
    public string BoxerToPutMoneyOn { get; set; }
    public void Update(IObservable subject)
    {
        var boxFight = (BoxFight)subject;
        if (boxFight.BoxerAScore < boxFight.BoxerBScore)
            BoxerToPutMoneyOn = "I put on boxer B, better be safe!";
        else BoxerToPutMoneyOn = "I put on boxer A, better be safe!";
    }
}
```

⁴¹ **Observer.** Intent. Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Спостерігач. Призначення. Визначає залежність один до багатьох між об'єктами таким чином, що коли один змінює свій стан, всі залежні є проінформовані та оновлені автоматично.

```
        Console.WriteLine("CONSERVATIVEPLAYER:{0}", BoxerToPutMoneyOn);  
    }  
}
```

Ось сама гра, яку споглядають наші гравці:

Уривок коду 19.2. Те, що споглядають – бій боксерів

```
interface ISubject  
{  
    void AttachObserver(IObserver observer);  
    void DetachObserver(IObserver observer);  
    void Notify();  
}  
class BoxFight : ISubject  
{  
    public List<IObserver> Observers { get; private set; }  
    public int RoundNumber { get; private set; }  
    private Random Random = new Random();  
  
    public int BoxerAScore { get; set; }  
    public int BoxerBScore { get; set; }  
  
    public BoxFight()  
    {  
        Observers = new List<IObserver>();  
    }  
    public void AttachObserver(IObserver observer)  
    {  
        Observers.Add(observer);  
    }  
    public void DetachObserver(IObserver observer)  
    {  
        Observers.Remove(observer);  
    }  
    public void NextRound()  
    {  
        RoundNumber++;  
  
        BoxerAScore += Random.Next(0, 5);  
        BoxerBScore += Random.Next(0, 5);  
  
        Notify();  
    }  
    public void Notify()  
    {  
        foreach (var observer in Observers)  
        {  
            observer.Update(this);  
        }  
    }  
}
```

Глянемо на *Спостерігача* в дії. Для цього створимо боксерський бій, який будуть споглядати гравці (не боксери звісно, бо вони б'ються і своє «бабло» вони «відкосять» у будь-якому випадку):

Уривок коду 19.3. Спостерігач в дії

```
var boxFight = new BoxFight();

var riskyPlayer = new RiskyPlayer();
var conservativePlayer = new ConservativePlayer();

boxFight.AttachObserver(riskyPlayer);
boxFight.AttachObserver(conservativePlayer);

boxFight.NextRound();
boxFight.NextRound();
boxFight.NextRound();
boxFight.NextRound();
```

А ось вивід на консоль:

```
RISKYPLAYER:I put on boxer A, if he win I get more!
CONSERVATIVEPLAYER:I put on boxer A, better be safe!

RISKYPLAYER:I put on boxer B, if he win I get more!
CONSERVATIVEPLAYER:I put on boxer A, better be safe!

RISKYPLAYER:I put on boxer B, if he win I get more!
CONSERVATIVEPLAYER:I put on boxer A, better be safe!

RISKYPLAYER:I put on boxer B, if he win I get more!
CONSERVATIVEPLAYER:I put on boxer A, better be safe!
```

20. Стан



Уявімо, що ми маємо розробити програму для опрацювання замовлень (*Orders*). Замовлення можуть бути в одному із декількох станів: новий (*NewOrder*), зареєстрований (*Registered*), погоджений (*Granted*), відправлений (*Shipped*), оплачений (*Invoiced*), відмінений (*Cancelled*).

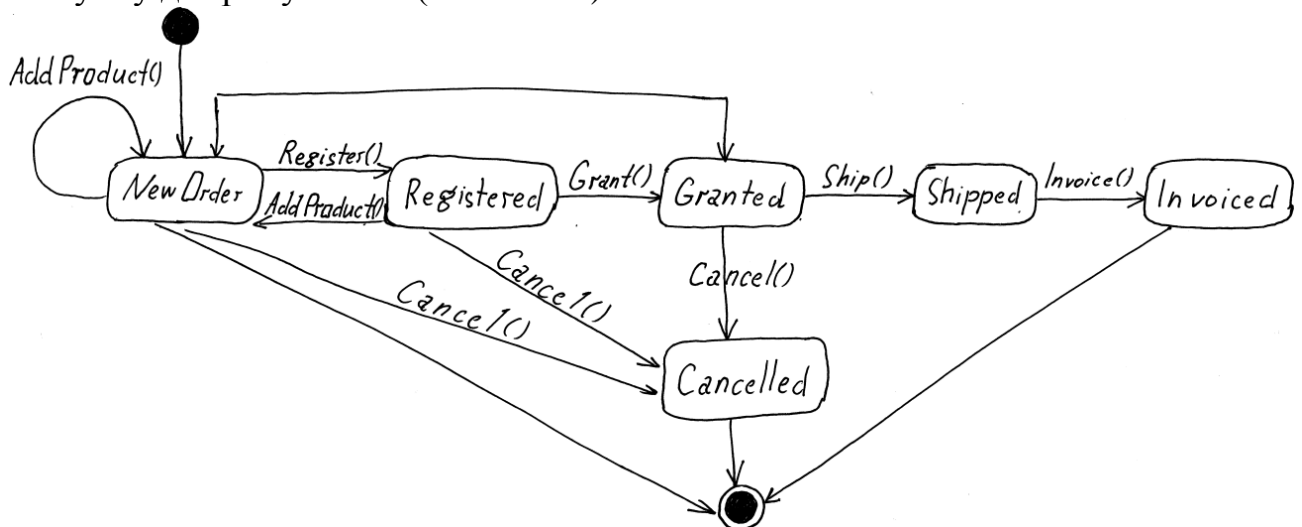
Також є певні правила, за якими замовлення може перейти в інший стан. Для прикладу, не можна відправити не зареєстроване замовлення.

Крім правил переходу є ще й інші правила, які визначають поведінку вашого замовлення. Наприклад, не можна додати продукт до замовлення тоді, коли воно є у відміненому стані.

Як можна гарно й чітко реалізувати таку систему поведінки замовлення?

Стан дозволяє винести логіку визначення стану об'єкту та його поведінку, характерну для цього стану, в інші класи.⁴²

Щоб поведінка замовлення і його станів була зрозуміла, глянемо на наступну діаграму станів (*state-chart*):

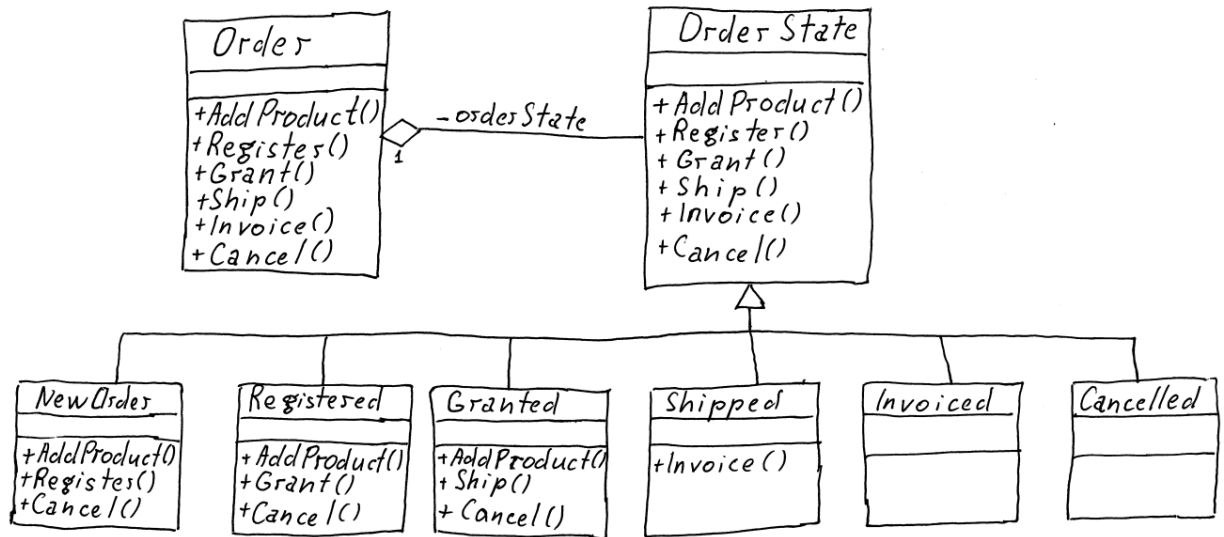


UML-діаграма 11. Діаграма станів замовлення

Ми можемо інкапсулювати поведінку, що пов'язана зі станом об'єкту в класах різних станів, що наслідуються від одного базового класу. Кожна із конкретних реалізацій буде відповідальна за надання можливості переходу з одного стану в інший.

⁴² **State.** Intent. Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Стан. Призначення. Дозволяє об'єкту змінити свою поведінку тоді, коли внутрішній стан змінюється. Буде здаватися, що об'єкт змінив свій клас.



UML-діаграма 12. Діаграма патерну Стан для нашого прикладу із замовленнями

І як же воно працює? Для початку зауважимо, що клас *Order* має поле-посилання на стан *_state*. Для того, щоб приклад виглядав правдоподібніше, добавимо також товари *_products*.

Уривок коду 20.1. Замовлення

```
class Order
{
    private OrderState _state;
    private List<Product> _products = new List<Product>();

    public Order()
    {
        _state = new NewOrder(this);
    }
    public void SetOrderState(OrderState state)
    {
        _state = state;
    }
    public void WriteCurrentStateName()
    {
        Console.WriteLine("Current Order's state: {0}", _state.GetType().Name);
    }
    // І так далі...
}
```

Уривок коду 20.2. В базовому класі *Order* делегує поведінку поточному стану

```
public void Ship()
{
    _state.Ship();
}
```

Наприклад, якщо поточний стан *Granted*, то метод *_state.Ship()* змінить стан замовлення на *Shipped* і якщо потрібно, зробить ще якусь специфічну для цього стану роботу.

Код нижче зображає конкретну реалізацію одного із станів. Конструктор базового класу містить параметер типу *Order*, що дозволяє стану містити поле-посилання на власника цього стану.

Уривок коду 20.3. Конкретна реалізація одного із станів

```
class Granted : OrderState
{
    public Granted(Order order)
        : base(order)
    {
    }
    public override void AddProduct()
    {
        _order.DoAddProduct();
    }
    public override void Ship()
    {
        _order.DoShipping();
        _order.SetOrderState(new Shipped(_order));
    }
    public override void Cancel()
    {
        _order.DoCancel();
        _order.SetOrderState(new Cancelled(_order));
    }
}
```

Якщо вас зацікавили методи класу *Order* на подібі *DoShipping()*, то в нашому прикладі вони імітують роботу, просто виводячи інформацію про дії з замовленням, але, звісно, ми вміщаємо там необхідну хитру логіку для виконання операцій, пов'язаних із поточним станом:

Уривок коду 20.4. Метод *DoShipping* для виконання загальної логіки

```
public void DoShipping()
{
    // Тут водій вантажівки нагужає ваше замовлення і «рулить»
    Console.WriteLine("Shipping...");
}
```

З іншого боку, логіку, яка стосується самого продукту, ми можемо виконувати у зовнішніх методах нашого замовлення не перевикликаючи її потім із стану, але це залежить від нас:

Уривок коду 20.5. Додавання продукту до замовлення

```
public void AddProduct(Product product)
{
    _products.Add(product);
    _state.AddProduct();
}
```

Якщо поточний стан *Registered*, то швидше за все такий стан не має перевизначеного методу *ship()*, а має тільки методи *addProduct()*, *grant()* та *cancel()*. Таким чином метод базового класу буде викликаний. *OrderState*, він же базовий клас, має всі методи, які можуть бути перевизначені у станах, але всі вони «плюються» експешинами, або ж просто виводять щось у консоль, як у нашому прикладі:

Уривок коду 20.6. Базовий клас стану замовлення

```
class OrderState
{
    public Order _order;

    public OrderState(Order order)
    {
        _order = order;
    }
    public virtual void AddProduct()
    {
        OperationIsNotAllowed("AddProduct");
    }
    // Наступні методи (Register, Grant, Ship, Invoice, Cancel) виглядають так же
    private void OperationIsNotAllowed(string operationName)
    {
        Console.WriteLine("Operation {0} is not allowed for Order's state {1}",
            operationName, this.GetType().Name);
    }
}
```

Здійсимо певний перелік операцій із створення замовлення, додаванням до нього нашого улюбленого пива і доставки на дім:

Уривок коду 20.7. Декілька операцій для демонстрації використання

```
Product beer = new Product();
beer.Name = "MyBestBeer";
beer.Price = 78000;

Order order = new Order();
order.WriteCurrentStateName();

order.AddProduct(beer);
order.WriteCurrentStateName();

order.Register();
order.WriteCurrentStateName();

order.Grant();
order.WriteCurrentStateName();

order.Ship();
order.WriteCurrentStateName();
```

Вивід:

```
Current Order's state: NewOrder
Adding product...
Current Order's state: NewOrder
Registration...
Current Order's state: Registered
Granting...
Current Order's state: Granted
Shipping...
Current Order's state: Shipped
Invoicing...
Current Order's state: Invoiced
Press any key to continue . . .
```


Давайте-но додамо ще трохи пивка до замовлення, яке нам вже відправили:

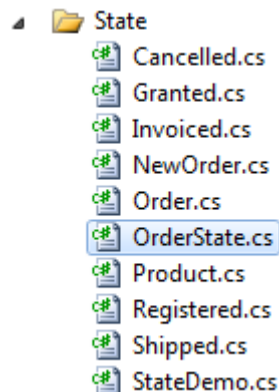
Уривок коду 20.8. Демонстрація неправильного використання замовлення і поведінки стану

```
// Пробуємо дозамовити пива до вже відправленого замовлення
// і дивимось що буде (див. довгий червоний рядок у виводі)
order.AddProduct(beer);
order.WriteCurrentStateName();
```

Вивід:

```
Current Order's state: NewOrder
Adding product...
Current Order's state: NewOrder
Registration...
Current Order's state: Registered
Granting...
Current Order's state: Granted
Shipping...
Current Order's state: Shipped
Operation AddProduct is not allowed for Order's state Shipped
Current Order's state: Shipped
Press any key to continue . . .
```

Одним із суттєвих недоліків цього дизайн патерну є розплід великої кількості класів станів:



Малюнок 4. Розплід великої кількості класів-станів

Але з іншої сторони, саме так ми можемо чітко розділяти поведінку в залежності від станів. Я читав про вирішення цієї проблеми за допомогою таблицки на зразок `[state/method/state]`, що зберігає дозволені переходи. Проблема також може бути вирішена за допомогою `switch` («мда», щось воно не звучить)!⁴³

⁴³ Гарно про еволюцію цих підходів можна прочитати в книзі Jimmy Nilsson "Applying Domain-Driven Design and Patterns".

21. Стратегія



Просто, як двері – якщо на дворі дощ, то ви берете парасольку і куртку, а якщо палить сонце, то ви берете футболку і сонцезахисні окуляри. Що вдягати є вашою стратегією, яку ви змінюєте залежно від обставин. Але замість того, щоб дивитися за вікно яка погода, а потім мандрувати до шафки із одягом і вибирати що одягнути, а далі, пам'ятаючи яка погода, йти до іншої шафки і брати парасольку або окуляри, ви просто піднімаєтеся із ліжка протираєте очі і вам у руки жінка (чоловік, що малоімовірно) подає потрібний одяг і аксесуари. Іншими словами, стратегія на сьогоднішній день просто була подана і ви нею скористалися.

Стратегія зберігає сім'ю алгоритмів і дозволяє змінювати їх незалежно та переключатися між ними.⁴⁴

Розглянемо два підходи до вирішення цієї проблеми. Припустимо, що в класі *Myself* ми маємо метод для походу на вулицю *GoOutside()* в якому ми вибираємо собі одяг і «шуруємо» на вулицю.

Уривок коду 21.1. Спосіб 1. Клас *Myself* із «мудрим» методом *GoOutside*

```
class Myself
{
    public void GoOutside()
    {
        var weather = Weather.GetWeather();
        string clothes = GetClothes(weather);
        string accessories = GetAccessories(weather);
        Console.WriteLine("Today I wore {0} and took {1}", clothes, accessories);
    }
    private string GetAccessories(string weather)
    {
        string accessories;
        switch (weather)
        {
            case "sun":
                accessories = "sunglasses";
                break;
            case "rain":
                accessories = "umbrella";
                break;
            default:
                accessories = "nothing";
                break;
        }
        return accessories;
    }
}
```

⁴⁴ **Strategy.** Intent. Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Стратегія. Призначення. Визначає сім'ю алгоритмів, інкапсулює кожен з них і робить їх взаємозамінними. Стратегія дозволяє видозмінювати алгоритм незалежно від клієнтського коду, що використовує стратегію.

```
private string GetClothes(string weather)
{
    string clothes;
    switch (weather)
    {
        case "sun":
            clothes = "T-Shirt";
            break;
        case "rain":
            clothes = "Coat";
            break;
        default:
            clothes = "Shirt";
            break;
    }
    return clothes;
}
```

Воно звичайно добре, але як тільки вам треба буде підлаштовуватися до снігу, що несподівано випав, вам прийдеться додати ще один *case* в трьохстах місцях. З однієї сторони це не складно, але з іншої, з часом це може вас «дістати». Також з часом код із методом *GoOutside()* вже не можна буде змінювати із-за певних причин. Що тоді?

Вище наводився приклад із *switch* для того, щоб показати, що *Стратегія* – це елегантний спосіб позбутися цього «чудіща».

Уривок коду 21.2. Спосіб 2. Вирішення проблеми за допомогою Стратегії

```
class Myself
{
    private IWearingStrategy _wearingStrategy = new DefaultWearingStrategy();

    public void ChangeStrategy(IWearingStrategy wearingStrategy)
    {
        _wearingStrategy = wearingStrategy;
    }
    public void GoOutside()
    {
        var clothes = _wearingStrategy.GetClothes();
        var accessories = _wearingStrategy.GetAccessories();
        Console.WriteLine("Today I wore {0} and took {1}", clothes, accessories);
    }
}
```

Як бачимо, ми маємо інтерфейс стратегії із двома методами. Глянемо, як виглядає стратегія на сонячний день:

Уривок коду 21.3. Сонячна стратегія

```
interface IWearingStrategy
{
    string GetClothes();
    string GetAccessories();
}
```

```
class SunshineWearingStrategy : IWearingStrategy
{
    public string GetClothes()
    {
        return "T-Shirt";
    }

    public string GetAccessories()
    {
        return "sunglasses";
    }
}
```

Все, що нам залишилося, то це правильно проставити стратегію. Ага! Все одно хтось буде змушений поставити правильну стратегію (жінка, яка встала раніше і глянула у вікно, це вона може зробити ще одним свічком, про який ви нічого не знаєте).

Уривок коду 21.4. Елегантне надання об'єкту класу *Myself* стратегії

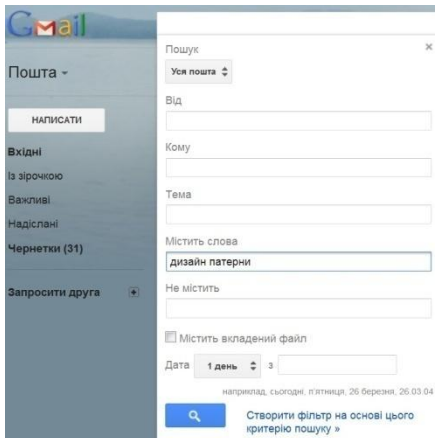
```
var me = new Myself();
me.ChangeStrategy(new RainWearingStrategy());
me.GoOutside();
```

Вивід простий:

```
Today I wore Coat and took umbrella
```

Ще одним (але не моїм) гарним прикладом є зміна стратегії сортування списку в залежності від його розміру. Всі ми знаємо що для малої кількості даних достатньо сортування вставками або якийсь інший простий спосіб. Для відносно великих списків найоптимальніше використовувати швидко сортування, а для дуже великої кількості даних – пірамідальне. Так от, алгоритм зберігається у своєму класі і в залежності від кількості елементів ми просто міняємо реалізацію алгоритму.

22. Шаблонний метод



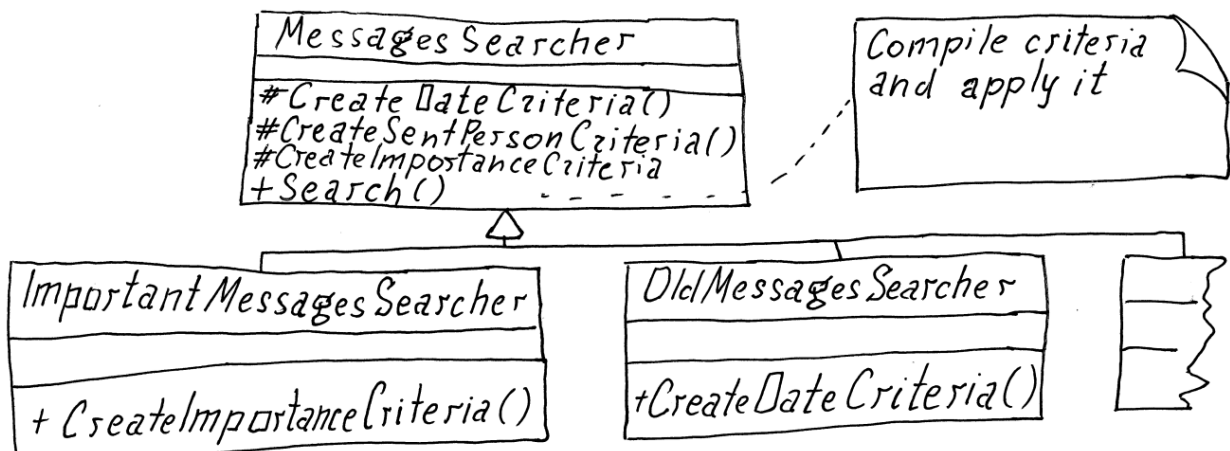
Уявімо собі, що ми маємо розробити систему пошуку повідомлень для поштової стриньки. Процес пошуку складається із декількох операцій, які є загальними для всіх повідомлень, але специфіка методів може відрзнятися для деяких типів повідомлень.

Вам слід написати клас *Searcher*, який буде інкапсулювати алгоритм пошуку, але ви також хочете залишити можливість перевизначити деякі елементи алгоритму для певних методів. Як це

можна легко зробити?

Шаблонний Метод задає покроково алгоритм, а елементи алгоритму можуть бути довизначені в похідних класах.⁴⁵

Патерн сам по собі є досить інтуїтивним, так само як і його реалізація. Вам потрібен базовий клас, що містить основні операції і один *Шаблонний Метод* (*Search*), який буде оперувати базовими операціями. Кожна із операцій може бути перевантажена в похідному класі.



UML-діаграма 13. Шаблонний Метод

Я написав тривіальну імплементацію патерну, оскільки викликаю операції одну за одною в шаблонному методі. В реальному житті, швидше за все, буде складний алгоритм, побудований на основі базових операції. І вам буде потрібно тільки перевантажити деякі із них, або, інакше кажучи, перевантажити частини алгоритму. Також можна позначити базові операції за допомогою *abstract*, що буде вимагати їхньої імплементації у похідних класах.

⁴⁵ **Template Method.** Intent. Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. **Шаблонний Метод.** Призначення. Визначає скелет алгоритму в операціях, передаючи деякі кроки підкласам. Шаблонний Метод дозволяє підкласам перевизначити певні кроки алгоритму без зміни структури алгоритму.

Уривок коду 22.1. Клас MessagesSearcher із шаблонним методом

```
class MessagesSearcher
{
    protected DateTime DateSent;
    protected String PersonName;
    protected int ImportanceLevel;

    public MessagesSearcher(DateTime dateSent, String personName, int
importanceLevel)
    {
        DateSent = dateSent;
        PersonName = personName;
        ImportanceLevel = importanceLevel;
    }

    // Базові операції (primitive operations)
    protected void CreateDateCriteria()
    {
        Console.WriteLine("Standard date criteria has been applied.");
    }
    protected void CreateSentPersonCriteria()
    {
        Console.WriteLine("Standard person criteria has been applied.");
    }
    protected void CreateImportanceCriteria()
    {
        Console.WriteLine("Standard importance criteria has been applied.");
    }

    // Метод, який називають шаблонним
    public String Search()
    {
        CreateDateCriteria();
        CreateSentPersonCriteria();
        Console.WriteLine("Template method does some verification accordingly to
search algo.");
        CreateImportanceCriteria();
        Console.WriteLine("Template method verifies if message could be so
important or useless from person provided in criteria.");
        Console.WriteLine();
        return "Some list of messages...";
    }
}

class ImportantMessagesSearcher : MessagesSearcher
{
    public ImportantMessagesSearcher(DateTime dateSent, String personName)
        : base(dateSent, personName, 3) // «3» означає, що повідомлення важливе
    {
    }

    // Одна операція перевантажена (IMPORTANT в кінці)
    protected void createImportanceCriteria()
    {
        Console.WriteLine(
            "Special importance criteria has been formed: IMPORTANT");
    }
}
```

```
class UselessMessagesSearcher : MessagesSearcher
{
    public UselessMessagesSearcher(DateTime dateSent, String personName)
        : base(dateSent, personName, 1) // «1» означає, що «в пень» воно треба
    {
    }
    // Одна операція перевантажена (вивід відрізняється словом «USELESS» в кінці)
    protected void createImportanceCriteria()
    {
        Console.WriteLine("Special importance criteria has been formed: USELESS");
    }
}
```

Уривок коду 22.2. Використання

```
MessagesSearcher searcher = new UselessMessagesSearcher(DateTime.Today, "Sally");
searcher.Search();

searcher = new ImportantMessagesSearcher(DateTime.Today, "Killer");
searcher.Search();
```

Вивід для першого та другого пошуку:

```
Standard date criteria has been applied.
Standard person criteria has been applied.
Template method does some verification accordingly to search algo.
Special importance criteria has been formed: USELESS
Template method verifies if message could be so important or useless from person
provided in criteria.
```

```
Standard date criteria has been applied.
Standard person criteria has been applied.
Template method does some verification accordingly to search algo.
Special importance criteria has been formed: IMPORTANT
Template method verifies if message could be so important or useless from person
provided in criteria.
```


23. Відвідувач



Уявімо собі, що ви нарешті спромоглися створити свою власну компанію, і оскільки вона пристойного розміру, ви вирішили орендувати для неї цілу будівлю. У нас держава дуже хороша і дбає про підприємства. А щоб у підприємств усе відповідало вимогам, постійно висилаються різноманітні перевірки. Причому правила, за якими перевіряють ваше підприємство, постійно змінюються.

Найближчим часом вам слід буде прийняти багато відвідувачів (*visitors*), таких як електрик (*electrician*), сантехнік (*plumber*), податківець і так далі... Усі вони будуть перевіряти вашу будівлю вздовж і в поперек, проходячи від поверху до поверху, від кімнати до кімнати. Я підозрюю, що якась певна схема класів у вас уже появилася у голові. Якщо так, то у мене є наступне питання: де має жити логіка певної перевірки будівлі? Чи має будівля знати, як перевіряти електричні щитки, чи це має знати електрик, або чи має знати кімната, як перевірити вимикачі, чи це так само робота електрика? Звичайно що електрик, який і є відвідувачем, інкапсулює логіку перевірки певних елементів (*elements*) вашої будівлі.

Відвідувач (Visitor) дозволяє відділити певний алгоритм від елементів, на яких алгоритм має бути виконаний, таким чином ми можемо легко додати або ж змінити алгоритм без змін до елементів системи. Як на мене, це і є однією із найбільш помітних переваг цього патерну.⁴⁶

Отже, як і було згадано вище, інкапсульована логіка живе у конкретному відвідувачі. Ця логіка може бути застосована до елементів системи. Основу цього дизайн патерну можна вибудувати на двох інтерфейсах. Ось вони:

Уривок коду 24.1. IVisitor та IElement

```
interface IVisitor
{
    void Visit(OfficeBuilding building);
    void Visit(Floor floor);
    void Visit(Room room);
}
interface IElement
{
    void Accept(IVisitor visitor);
}
```

⁴⁶ **Visitor.** Intent. Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Відвідувач. Призначення. Представляє операцію, яка має бути виконана на елементах структури об'єкта. Відвідувач дозволяє визначити нову операцію без зміни класів елементів-операндів.

У нашому прикладі *ElectricitySystemValidator* є однією із конкретних реалізацій інтерфейсу *IVisitor*, яка може виглядати як наведений нижче код:

Уривок коду 24.1. Один із відвідувачів – електрик

```
class ElectricitySystemValidator : IVisitor
{
    public void Visit(OfficeBuilding building)
    {
        var electricityState = (building.ElectricitySystemId > 1000)
                                ? "Good" : "Bad";

        Console.WriteLine(
            string.Format("Main electric shield in building {0} is in {1} state.",
                building.BuildingName, electricityState));
    }
    public void Visit(Floor floor)
    {
        Console.WriteLine(
            string.Format("Diagnosing electricity on floor {0}.",
                floor.FloorNumber));
    }
    public void Visit(Room room)
    {
        Console.WriteLine(
            string.Format("Diagnosing electricity in room {0}.", room.RoomNumber));
    }
}
```

Про що говорить нам цей клас? Ми зауважили, що у відповідності до інтерфейсу *IVisitor* в одному відвідувачі є три методи, кожен із яких описує логіку перевірки для одного із елементів. Виходячи із цього, ми можемо з чистою совістю проводити нашого відвідувача із поверху на поверх та з кімнати до кімнати.

Клас *PlumbingSystemValidator* схожий на *ElectricitySystemValidator*, але в своїй логіці бере до уваги вік будівлі, щоб приблизно оцінити на скільки сантехнічна частина справна. Що ще цікаво про цей клас, так це те, що він нічого не робить у кімнатах. Звісно, якщо ваше підприємство є якимось хім-заходом, сантехніку доведеться пройтися по всіх кімнатах.

До цього часу вже стало зрозуміло, що структура будівлі обхідна. Все починається із будівлі (*OfficeBuilding*), яка має поверхи (*Floors*), і кожен із поверхів може мати багато кімнат. Глянемо на імплементацію поверху.

Уривок коду 24.1. Один із елементів будівні – поверх

```
class Floor : IElement
{
    private readonly IList<Room> _rooms = new List<Room>();
    public int FloorNumber { get; private set; }
    public IEnumerable<Room> Rooms { get { return _rooms; } }

    public Floor(int floorNumber)
    {
        FloorNumber = floorNumber;
    }
}
```

```
    }  
    public void AddRoom(Room room)  
    {  
        _rooms.Add(room);  
    }  
    public void Accept(IVisitor visitor)  
    {  
        visitor.Visit(this);  
        foreach (var room in Rooms)  
        {  
            room.Accept(visitor);  
        }  
    }  
}
```

Як можна побачити, цей клас містить метод *Accept*, який вимагається інтерфейсом і який приймає відвідувача. В середині цього методу ми виконуємо наш алгоритм і, якщо треба, передаємо нашого відвідувача «по колу». Як бачимо, ніякі технічні перевірки не виконуються напряму у цьому класі, тому ми можемо бути певні, що якщо у майбутньому слід буде змінити спосіб перевірки електросистеми у кімнаті, то це буде зроблено у відвідувачі без будь-яких впливів на клас кімнати.

OfficeBuilding є досить подібним класом, хіба що має багато інших додаткових властивостей. *Room* взагалі є простим класом, який не агрегує чи компонує інших елементів.

Глянемо на код використання дизайн патерну:

Уривок коду 24.1. Маємо будівлю із 2-ма поверхами, на кожному є по 3 кімнати. Запускаємо у будівлю електрика і сантехніка як відвідувачів

```
var floor1 = new Floor(1);  
floor1.AddRoom(new Room(100));  
floor1.AddRoom(new Room(101));  
floor1.AddRoom(new Room(102));  
var floor2 = new Floor(2);  
floor2.AddRoom(new Room(200));  
floor2.AddRoom(new Room(201));  
floor2.AddRoom(new Room(202));  
var myFirmOffice = new OfficeBuilding("[Design Patterns Center]", 25, 990);  
myFirmOffice.AddFloor(floor1);  
myFirmOffice.AddFloor(floor2);  
  
var electrician = new ElectricitySystemValidator();  
myFirmOffice.Accept(electrician);  
  
var plumber = new PlumbingSystemValidator();  
myFirmOffice.Accept(plumber);
```

Вивід:

```
Main electric shield in building [Design Patterns Center] is in Bad state.  
Diagnosing electricity on floor 1.  
Diagnosing electricity in room 100.  
Diagnosing electricity in room 101.
```

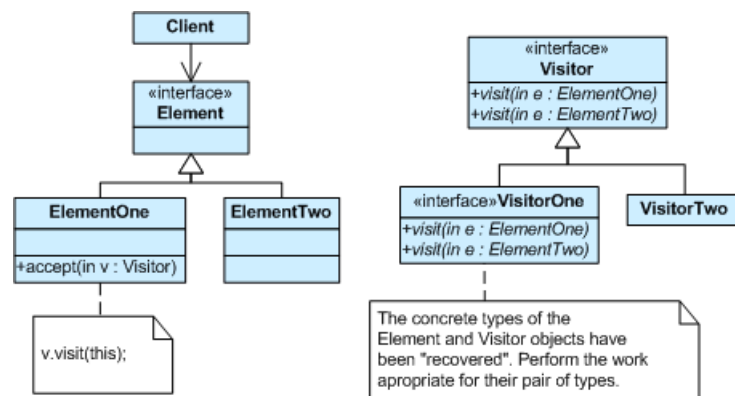
```

Diagnosing electricity in room 102.
Diagnosing electricity on floor 2.
Diagnosing electricity in room 200.
Diagnosing electricity in room 201.
Diagnosing electricity in room 202.
Plumbing state of building [Design Patterns Center] probably is in Good condition,
since building is New.
Diagnosing plumbing on floor 1.
Diagnosing plumbing on floor 2.

```

UML діаграма класів

Чим більше я пишу про дизайн патерни, тим більше я розумію що UML діаграми часто можуть ввести в оману. Ті діаграми, що представлені в GoF книжці насправді хороші, але вони зображають одні із найбільш частих випадків застосування певного патерну. Таким чином, *Відвідувача* найчастіше зображають як один базовий клас із двома похідними. Базовий клас визначає, що похідні мають реалізовувати «відвідування» елементів системи. Елемент системи може мати декілька реалізацій також.



UML-діаграма 14. Стандартна діаграма Відвідувача⁴⁷

Але коли я собі думаю, як інакше цей дизайн патерн може бути реалізований, то діаграми будуть виглядати зовсім інакше. Зокрема, із використанням *Компонувальника*, можна реалізувати тільки один клас для елемента, який є деревовидною структурою і містить вкладені елементи. Або ваша програма може містити один клас для відвідувача і один для елемента. За цим криється патерн, але не так і просто його розгледіти. Хоча й навіщо?...

Однією із переваг патерну є те, що він відокремлює алгоритм від елементів, до яких він має бути застосований, але це одночано й недолік, оскільки інтерфейс елемента має бути досить розвинений для нормальної роботи *Відвідувача*.⁴⁸

Ще одним недоліком є порушення зв'язності системи, оскільки із використанням цього патерну прийдеться добавляти велику кількість методів в елементи, які будуть відвідуватися.⁴⁹

⁴⁷ Взято із: http://sourcemaking.com/design_patterns/visitor

⁴⁸ Як було згадано Геннадієм Омельченко в коментарі на блозі .NET User Group.

Використані матеріали та подальші рекомендації

Цю сторінку я додаю тільки тому, що будуть звучати незадоволені вигуки на зразок «А де список літератури?». Основою всього написаного у цій книзі є книга GoF⁵⁰ та мій досвід. Звичайно, я перечитав багато різних статей на різних сайтах, спробував різні програми для UML, знайшов різні картинки для патернів. Зазвичай в мене є згадки із посиланнями на першоджерела в зносках.

Не знаю чому ви брали до рук цю книгу. Якщо для того, щоб почитати щось цікавеньке про те, що вже й так знаєте, тоді рекомендацій по вивченню патернів, мабуть не даватиму – ви й так знаєте, за що наступне братися. Якщо ви дійсно читали книгу, щоб ознайомитися із патернами, то було б чудово, якби ви прямо зараз сіли за комп'ютер і придумали для себе завдання, в якому можна буде використати декілька патернів. Або напишіть *Відвідувача* по пам'яті – прочитайте із книги проблему, яка вирішується і означення і починайте програмувати. Головне, вирішуйте проблему, а патерн в кінці-кінців вималюється. Звичайно, що я б рекомендував прочитати оригінальну книгу, але одним читанням багато не досягнеш. Як бачите, я сам для себе вирішив, що реалізую усі GoF дизайн патерни, за одно і книгу написав. Чому б вам не придумати щось цікаве для вивчення патернів або чогось іншого? Вперед!

⁴⁹ Як підмічено Андрієм в коментарі на блозі .NET User Group.

⁵⁰ Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Про автора

Андрій Будаї почав працювати як програміст-інженер, ще до того як закінчив університет. Вже за декілька місяців після отримання магістерського диплому він отримав позицію «сініор» інженера. Опісля він також виконував роль тех-ліда на проекті. Зараз він успішний і перспективний .NET програміст. До тепер, починаючи із 2008 він розробляв і дизайнив рішення на платформі .NET. Він дуже цілеспрямований і відкритий розробник, що прагне вчити нові технології. Ви можете запросто впіймати його онлайн на його блозі <http://andriybuday.com/> або просто почати слідувати за ним на twitter [@andriybuday](https://twitter.com/andriybuday).



Буду радий почути відгуки!

<http://designpatterns.andriybuday.com/>

<http://andriybuday.com/>

andriybuday@gmail.com