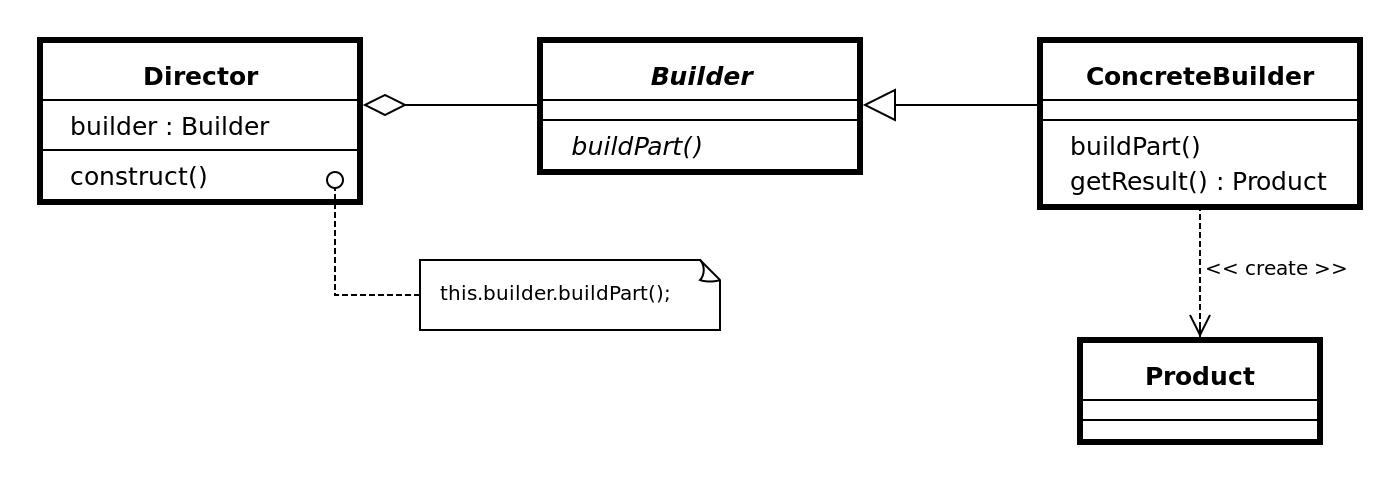
# Design Patterns Homework

Здравей, колега/колежке!

Реших моето домашно да се състои от по 1 design pattern от всяка от групите – creational, structural и behavioral. Също така реших да си избера най-интересните и съответно най-трудните, за да има някакъв смисъл от това домашно. Надявам се, че при съответната проверка ще ти помогне да разбереш някой от тях, ако все още имаш неясноти. Започваме!

# Builder Pattern

Като част от групата creational design patterns, Builder-а се занимава със създаване на обекти. За разлика от Factory Method-а и Abstract Factory-то целта му не е да „задейства“ полиморфизма, а да се пребори с т.нар. telescoping constructor anti-pattern, който се появява при наличието на много на брой конструктори с различни параметри. Най-просто казано Builder-а се използва при създаването на комплексни обекти или когато се налага създаването на обект в няколко стъпки. Постига се чрез разделянето на самия процес на конструиране и данните за създаването на съответния обект. Обяснение по самата диаграма би била доста по-лесно разбираема, за това ето я и нея.

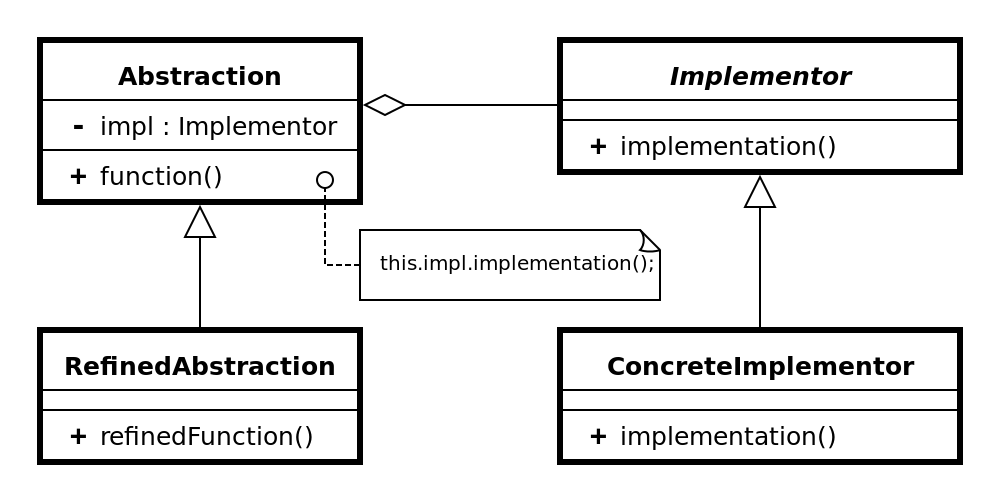


Имаме т.нар. Builder. В този клас (абстрактен или интерфейс) се дефинират стъпките за създаване на даден обект. В Director-а пък се дефинира точно в каква последователност се изпълняват тези стъпки. По-точно това става в метода construct. Там се извикват съответните стъпки в даден ред. ConcreteBuilder е наследник на Builder и той е същинския имплементатор на тези стъпки. В крайна сметка получаваме готовия продукт. Накратко казано – Director-а използва конкретна имплементация на Builder, чрез който се създава обекта. Това позволява много лесно добавяне на нови продукти, които имат същата структура, изпълнена по същите стъпки, но различни данни.

От демото, което съм написал (прилича много на това от лекциите, но примера много ми хареса) много лесно се вижда кой клас кое използва и какво се получава. За това и съм кръстил променливите в класа Program с имената от диаграмата. Много ясно се вижда, че само със създаването на нов ConcreteBuilder се получава съвсем различен краен продукт, което е и целта на pattern-а. Ако е необходимо създаването на обектите в различен ред, създава се нов Director и се използва той.

# Bridge Pattern

Bridge Pattern-а има на пръв поглед доста странно звучаща цел – да разграничи абстракцията от имплементацията, за да могат двете да се променят независимо. От тук нататък ще говоря на прост език, защото и на мен самия ми е трудно да синтезирам професионално звучащи изречения. Идеята на този pattern е да може поведението на даден клас да зависи от друг клас, т.е. на един и същ клас да можем да сложим напълно различни поведения без да променяме самия клас. От друга страна този pattern ни позволява и да използваме конкретната имплементация в други абстракции. Продължаваме по диаграмата на pattern-а.

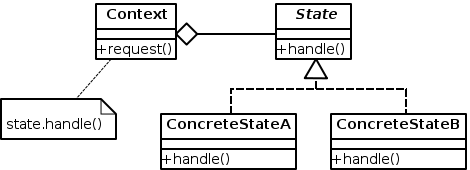


„да разграничи абстракцията от имплементацията“ – абстракцията се намира в класа Abstraction, а имплементацията в класа Implementor и по-конкретно в метода implementation. В абстракцията се пази инстанция на Inplementator-а, която се извиква, когато се наложи абстракцията да има някакво поведение.

Измислих едно демо, което може да не е връх на сладоледа, но подчертава използването на pattern-а и смятам, че е сравнително добре. В него имаме абстракцията „Нещо, което може да работи като часовник“ (класа Clock), и имплементатора ITimeFormatter, който има метод Format, който форматира даден час по определен начин. Абстракцията си има свои наследници, които си имат и собствено поведение, но и общото – да показват часа. Имаме и два начина на форматиране – европейският и американският. От примера се вижда колко лесно може да се добави нов начин на форматиране на часа – само със създаването на един прост клас с един метод и подпъхването му в конструктора на „часовника“. Точно това е идеята на pattern-а – runtime да може да се променя поведението на дадена абстракция и вместо да се прави огромна структура от класове и да се получи class explosion се прилага този pattern. Decorator pattern-а също може да реши подобен проблем, но той е по-скоро предназначен за добавяне на функционалност към класове чийто код нямаме, а Bridge е предназначен да се използва, когато всичко си пишем ние и искаме plugability.

# State Pattern

State pattern-а се използва, за да се позволи на обект да променя поведението си в зависимост от вътрешното си състояние. Използва се много често, когато има обект с няколко вътрешни състояния и смяната им трябва да става автоматично. Самия pattern не е от най-сложните, но според мен е много полезен и интересен. Съответно и диаграмата му не е сложна.



Обяснението на диаграмата е доста кратко. Имаме State, който дефинира действията зависещи от състояние. Имаме и ConcreteState-ове, които имплементират съответните действия. Context-а е класа, който има вътрешни състояния и в него се пази инстанция на текущия State. Когато трябва да се изпълни някой метод на Context-а и той зависи от вътрешното състояние, то се извиква същия метод на State-а.

За съжаление не успях да измисля пример, който да ми хареса повече от дадения ни на лекцията, за това прилагам него. Хубавото при него е, че е много интуитивен и лесно показва идеята на pattern-а. Най-интересната част е създаването на нов state с подаване на текущия. Но пък се сетих за лесен за разбиране и естествено елементарен за имплементация пример от реалния живот. Ако си представим една врата – в най-минималистичния вариант тя има 2 състояния: отворена и затворена. Когато вратата е затворена не можем да я затворим, а само да я отворим и обратно. В този случай метода Open при OpenState ще дава грешка, че не може да се извърши действието, а метода Close ще променя State-а на ClosedState. Това естествено е много елементарен пример, но е много лесен за разбиране.