Внедрение зависимостей

Первый пост в публичном пространстве группы, попробую сделать его коротким (поскольку длинные писать не умею) и полезным. Речь пойдет о механизме dependency injection и переломе моего мозга. Сам шаблон ныне очень популярен особенно в энетерпрайзной среде, но я даже с довольно большим стажем программирования очень не с первого раза въехал в то, что это такое. После прочтения довольно обширной статьи в википедии на ум приходит только:

Я понял - это намёк,
Я всё ловлю на лету
Но непонятно
Что конкретно ты имела в виду

Группа "Несчастный случай"

В смысле, все вроде правильно, но где тут вся суть-то?

Идея такая: предположим у нас есть какой-то объект, которому для работы нужны другие объекты. Скажем, у нас есть транспортный код, естественно для проведения моделирования, нам нужны такие вещи, как геометрический трэккер, генератор случайных чисел и симулятор взаимодействий. Мы не разбираем случай, когда все это делается на уровне субрутин, и перемешано в одну большую кучу тыщ на 10 строк кода. Так давно уже никто не делает (ну ладно, про никто это я погорячился).

> Внедрение или обращение зависимостей вообще имеет смысл только в объектном подходе, так что и говорим о нем, разумеется объектом может быть и функция.

Теперь подумаем, а каким образом наш транспортный код будет генерировать эти вспомогательные объекты.

1. Навсегда зашитым в коде способом. Если вдруг надо что-то поменять, лезем в код и перекомпиллируем. Если вдруг есть два различных кода для трэккинга, путаемся. Если вдруг вызов кода трекинга происходит не в одном месте, а в двух, трех, пяти и так далее, попадаем в ад той же степени (поменяли ссылку в одном месте и не поменяли в другом счастливо отлавливаем ошибку в течение пары месяцев). В программе больше, чем на пару сотен строк лучше так не делать.
2. Интерфейсом. Создаем интерфейс, описывающий набор поведений нужного нам класса. Реализуем этот интерфейс одним или несколькими способами (да, часто имеет смысл делать интерфейс с ровно одной реализацией). Далее мы можем получить объект, реализующий этот интерфейс разными способами:
    a. Генерировать его на ходу, используя конфигурацию материнского класса. Это весьма распространенный способ, но главная проблема в том, что при этом надо поддерживать весьма развесистую конфигурационную структуру, в которой легко запутаться. Если вдруг надо добавить для дочернего класса какую-нибудь дополнительную степень свободы, то ее надо будет протаскивать через всю структуру, что довольно затруднительно, если структура сложная.
    b. Определять нужные объекты в конструкторе материнского класса. То есть сначала создать трекер и генератор случайных чисел, а потом из них собрать транспорт. Проблема тут в том, что если таких элементов нужно 10, то конструктор превращается в сущего монстра. В таких случаях довольно успешно используются всяческого рода builder-ы.
    c. Сделать внешний сервис, который будет по запросу нашего материнского класса (транспорта) генерировать нужные элементы (трекер и так далее). При изменении конфигурации мы можем модифицировать только этот сервис, и ничего более.

Собственно последний вариант и принято называтьdependency injection. Вообще, разумеется вариант выглядит привлекательно. Дополнительный плюс в том, что так как сервис выдает объект по запросу, то генерация этого объекта, вообще говоря, может быть сделана в тот момент, когда он нужен, что может существенно оптимизировать время запуска программы. Каким образом эту довольно простую идею можно вытащить из того, что написано в википедии, я до сих пор не очень понимаю (особенно учитывая очень странное название).

А теперь к тому, с чего все это началось. Я сижу, изучаю тут код одних товарищей, которые очень любят этот самыйdependency injection, и используют его вообще везде. При этом эксплуатируется очень ныне популярная библиотека Guice. Я сперва, читая их код, вообще ничего не понял, поскольку в коде вообще нет никаких запросов к сервису, управляющему зависимостей. Оказывается, guice занимается тем, что при создании объекта, проводит поиск всех полей этого объекта, помеченных специальной аннотацией (в Java и всяческих наследниках, аннотации кода - это своеобразная метапрограмма). После чего, система ищет в себе генератор для объектов того типа, который нужен клиенту и подставляет его. В процессе писания этого поста, я наконец понял, как оно работает и после этого, все кажется довольно изящным, но перед этим мозг был сломан.

Вообще, интерес кdependency injection у меня возник до этого. Дело в том, что при разработке контекстов для DataForge, я сам того не зная, эту концепцию переизобрел. Одна из функций контекста - это как раз "внедрение зависимостей", он тащит в себе набор некоторых сервисов, которые могут быть вызваны любыми клиентами, знающими об этом контексте. Конечно, там еще много всяческих прелестей вроде наследования и динамического переключения, но в смысле архитектуры они вторичны.