**Flyweight Design Pattern:**

1. **Причина**:

* Използва споделяне, за да поддържа голям брой от малки обекти ефективно
  + Намаля необходимата памет за голям брой обекти
  + Споделя обекти, позволявайки да бъдат използвани в множество контексти едновременно
  + Запазва обектно ориентираната детайлност и гъвкавост
* Намаля употребата на памет, чрез споделяне колкото се може повече данни със други подобни обекти

2. **Проблем**:

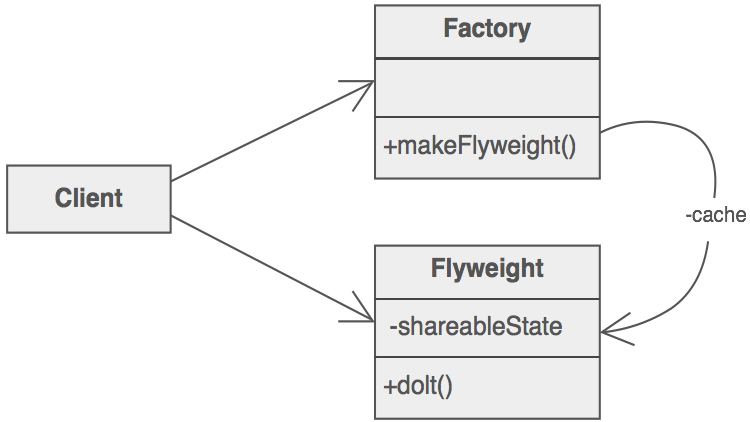
Проектирането на обекти до най-ниското ниво на детайлност предоставя оптимална гъвкавост, но може да бъде неприемливо скъпо в смисъл на работа и употреба на памет.

3. **На кратко**:

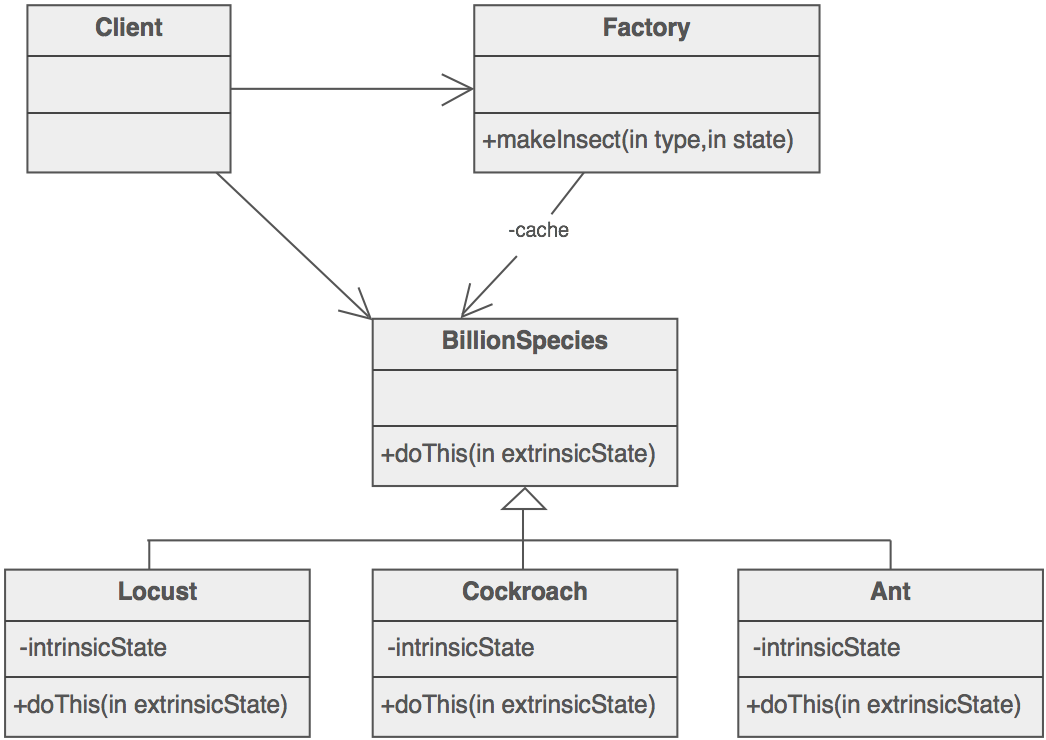
Flyweight модела описва как да се споделят обекти, което позволява тяхната употреба с висока детайлност, но без това да бъде на висока цена. Всеки flyweight обект е разделен на две части: зависимата от състоянието част и независимата от състоянието. Независимата се запазва/споделя в Flyweight обекта. Зависимото състояние се запазва или изчислява от клиентските обекти и се подава на Flyweight при извикване на операциите.

4. **Структура:**

Flyweight обектите се съхраняват в Factory хранилище. Клиента не създава Flyweight обект директно, а прави заявка към Factory. Всеки атрибут, който би направил споделянето невъзможно трябва да бъде предоставен от клиента, когато се направи заявка към Flyweight обект.



Скакалеца, хлебарката и мравката могат да бъдат ‚лека категория‘, защото състоянието на тяхната инстанция е деенкапсулирано или изкарано и трябва да бъде подадено от клиента.



**Adapter Design Pattern:**

1. **Причина**:

* Прехвърля интерфейса на клас в друг интерфейс, който клиентите очакват. Адаптер позволява класове да работят заедно, дори в случай когато те немогат заради несъвместими интерфейси.
* Обвива съществуващ клас с нов интерфейс.

2. **Проблем**:

Стар компонент предлага функционалност, която бихте искали да преизползвате, но има несъвместимост с архитектурата и философията на текущата система.

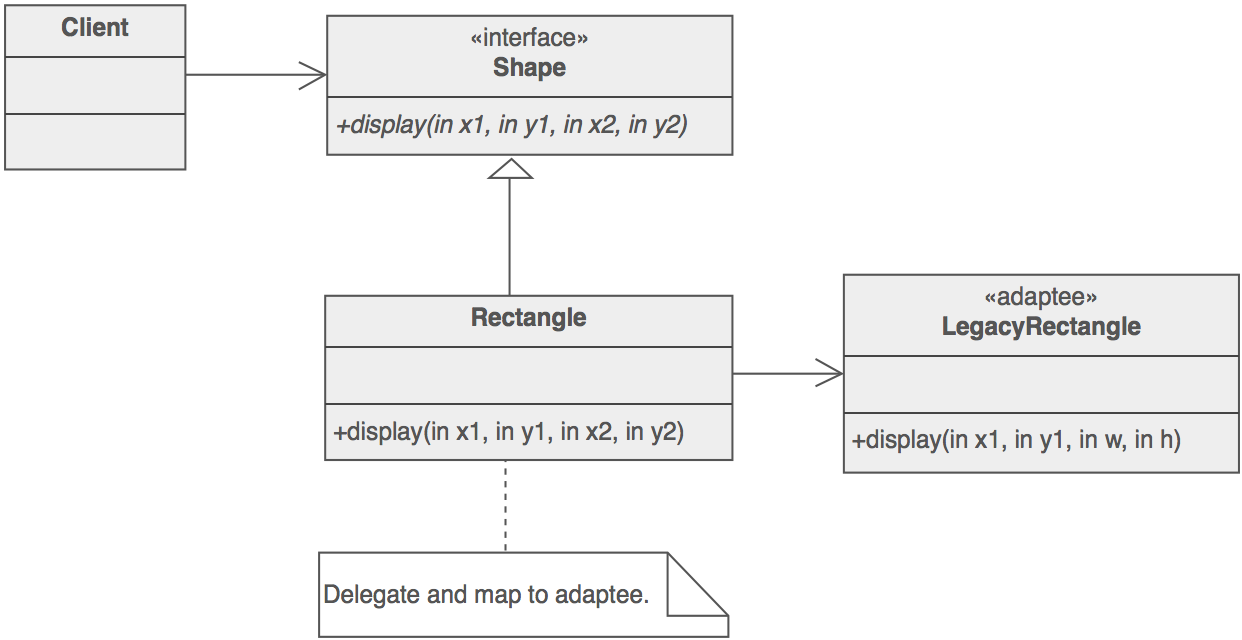
3. **На кратко**:

Адаптерът служи като посредник абстракция, която превежда стария компонент към новата система. Клиентите извикват методите на адаптера, които ги пренасочват към стария компонент. Тази стратегия може да бъде имплементирана или чрез наследяване или със добавяне.

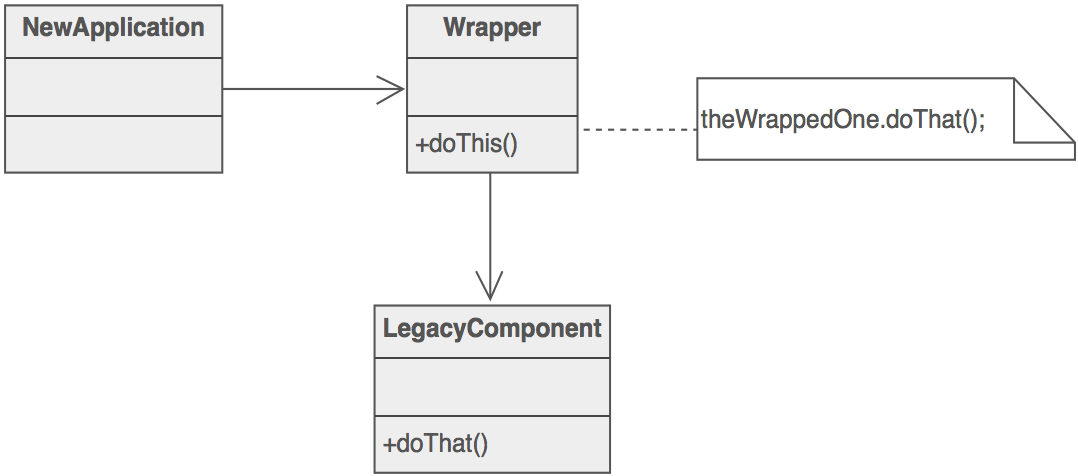
Адаптерът функционира като обвивка или модификатор на съществуващ клас. Предоставя различен или преведен изглед на класа.

4. **Структура:**

Старият метод на Rectangle компонента, display() очаква да приеме x,y,w и h параметри. Но клиента иска да подаде горен ляв “x“ и „y“, и долен десен „x“ и „y“. Това несъответсвие може да бъде променено, чрез добавяне на допълнително ниво на индиректност – Адаптер.



За адаптера също може да се мисли като за обвивка.



**Decorator Design Pattern:**

1. **Причина**:

* Добавяне на нови отговорности на обект динамично. Декораторите предоставят гъвкава алтернатива на създаването на допълнителни подкласове за разширяването на функционалност.
* Разкрасяване спрямо желанието на клиента, на основния обект, чрез рекурсивно обвиване.
* Обвиване на подаръка, слагане на подаръка в кутия и опаковане на кутията.

2. **Проблем**:

Искаш да добавиш поведение или състояние на индивидуални обекти по време на изпълнение на програмата.

3. **Структура:**

Клиента се интересува винаги в CoreFunctionality.doThis(). Клиента може да интересува от OptionalOne.doThis() и OptionalTwo.doThis(). Всеки от тези класове винаги делегира до Декоратора, а той винаги делегира до то обекта, на който е обвивка.

