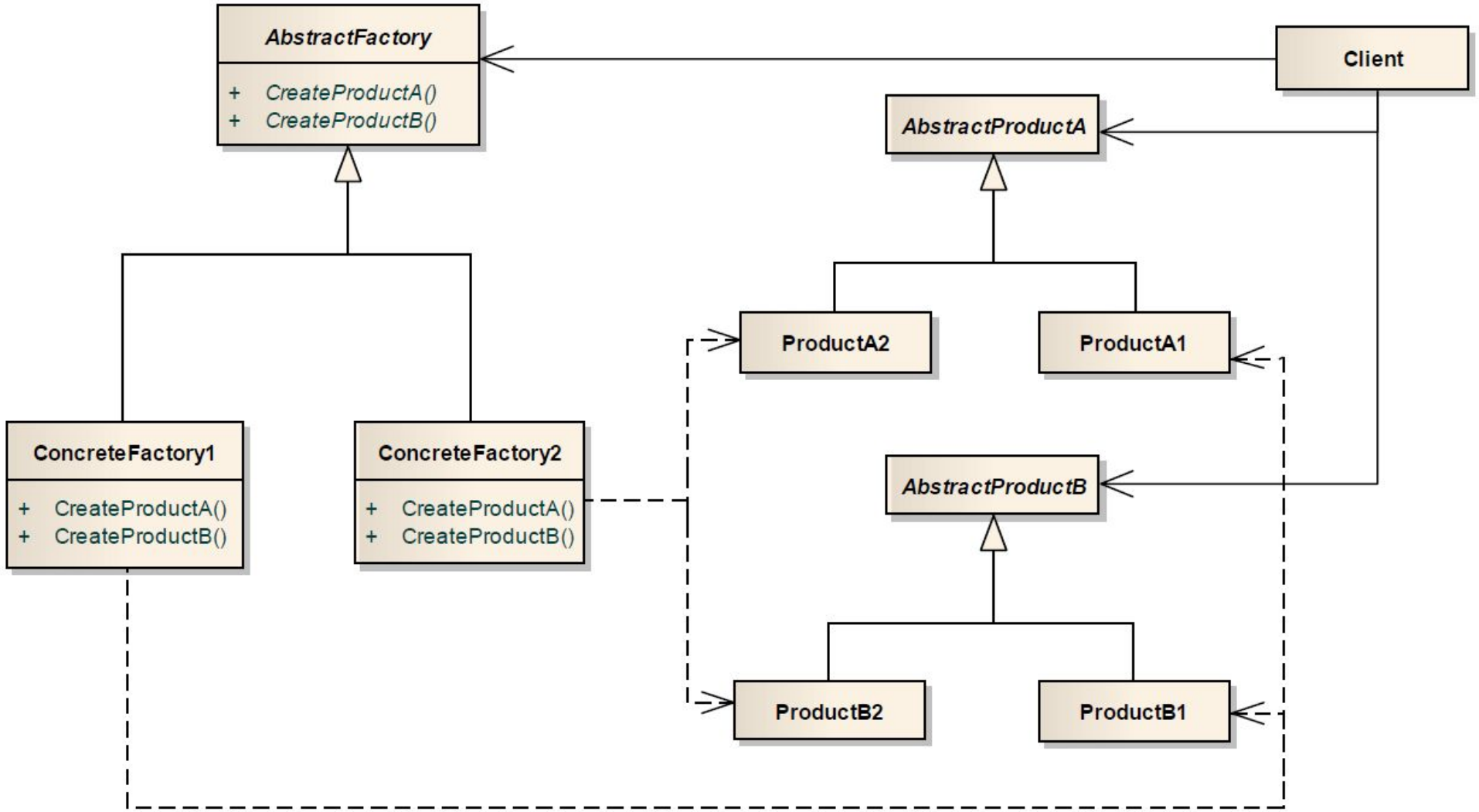
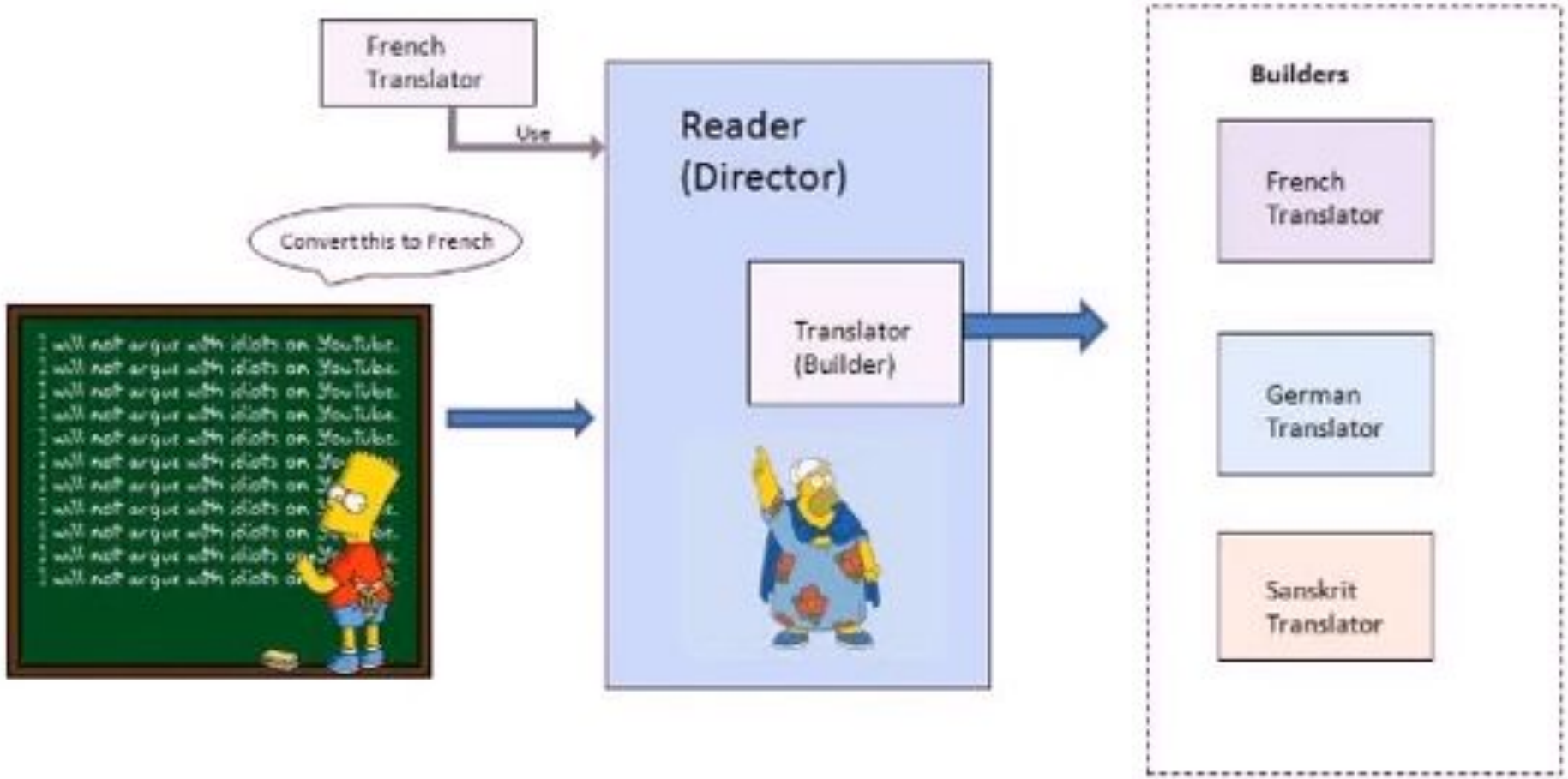
Паттерн Builder (строитель)

Идрисова И. А.

... Абстрактная фабрика



Зачем


French Translator
Use
Reader (Director)
Convertthis to French
Translator (Builder)
Builders
French Translator
German Translator
Sanskrit Translator

Описание паттернов проектирования

1. Название и классификация
2. Назначение
3. Псевдоним
4. Мотивация
5. Применимость
6. Структура
7. Участники
8. Отношения
9. Результаты
10. Реализация
11. Пример кода
12. Известные применения
13. Родственные паттерны

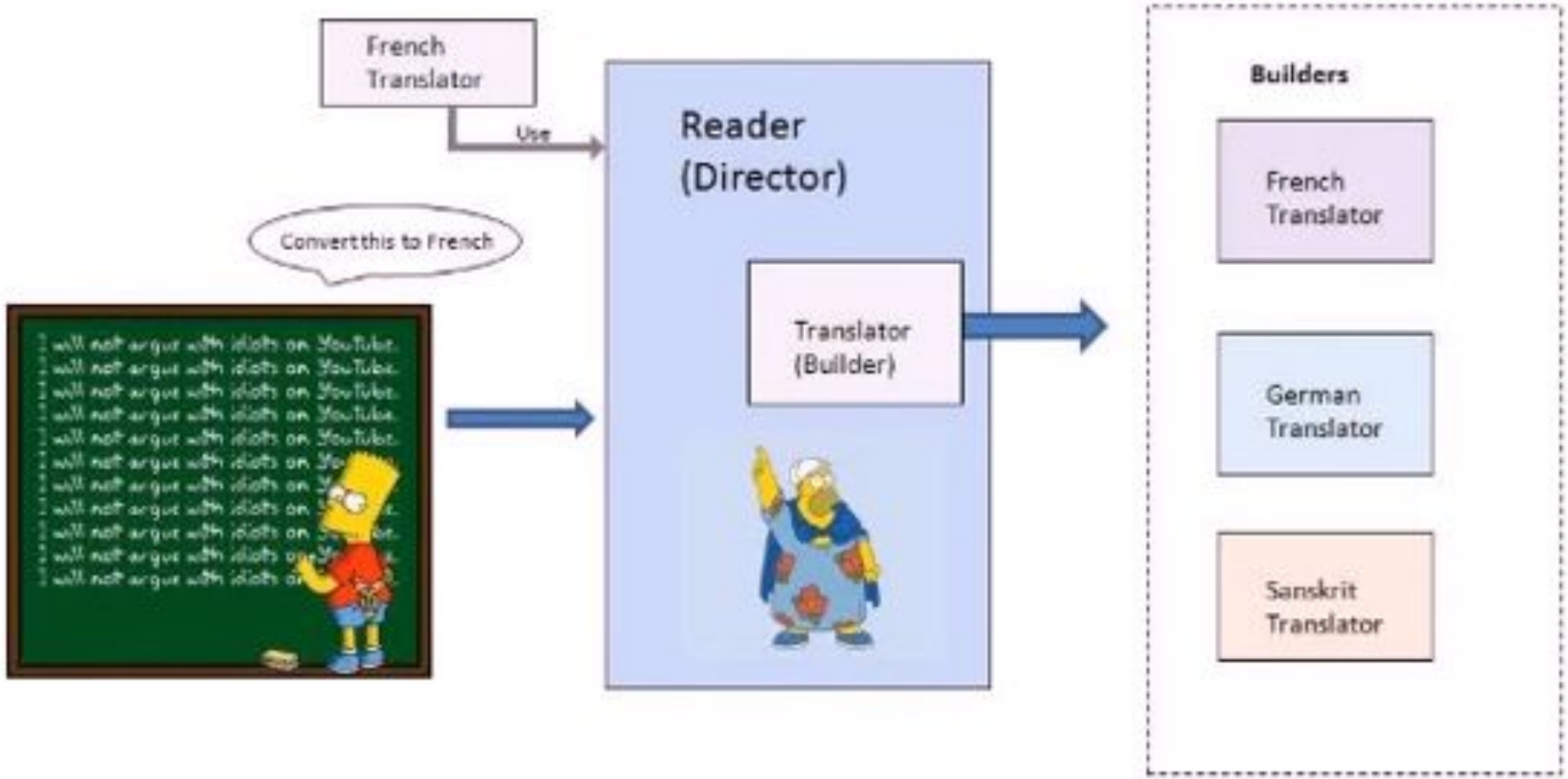
Название. Назначение. Псевдоним

Название и классификация
Строитель. Порождающие паттерны

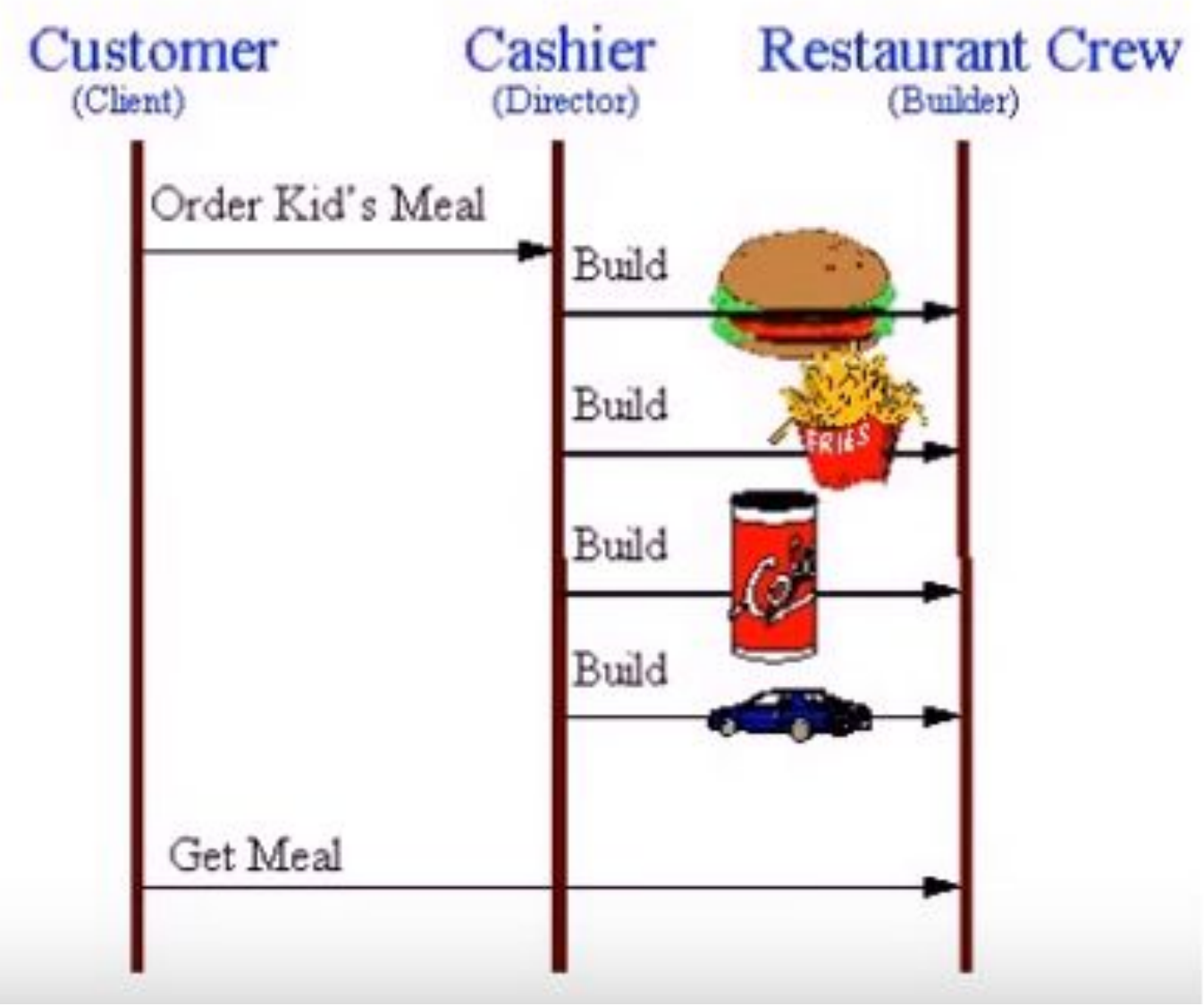
Назначение
Отделяет конструирование сложного объекта от его представления, так что
в результате одного и того же процесса конструирования могут получаться разные представления.

Псевдоним
строитель - просто строитель =)

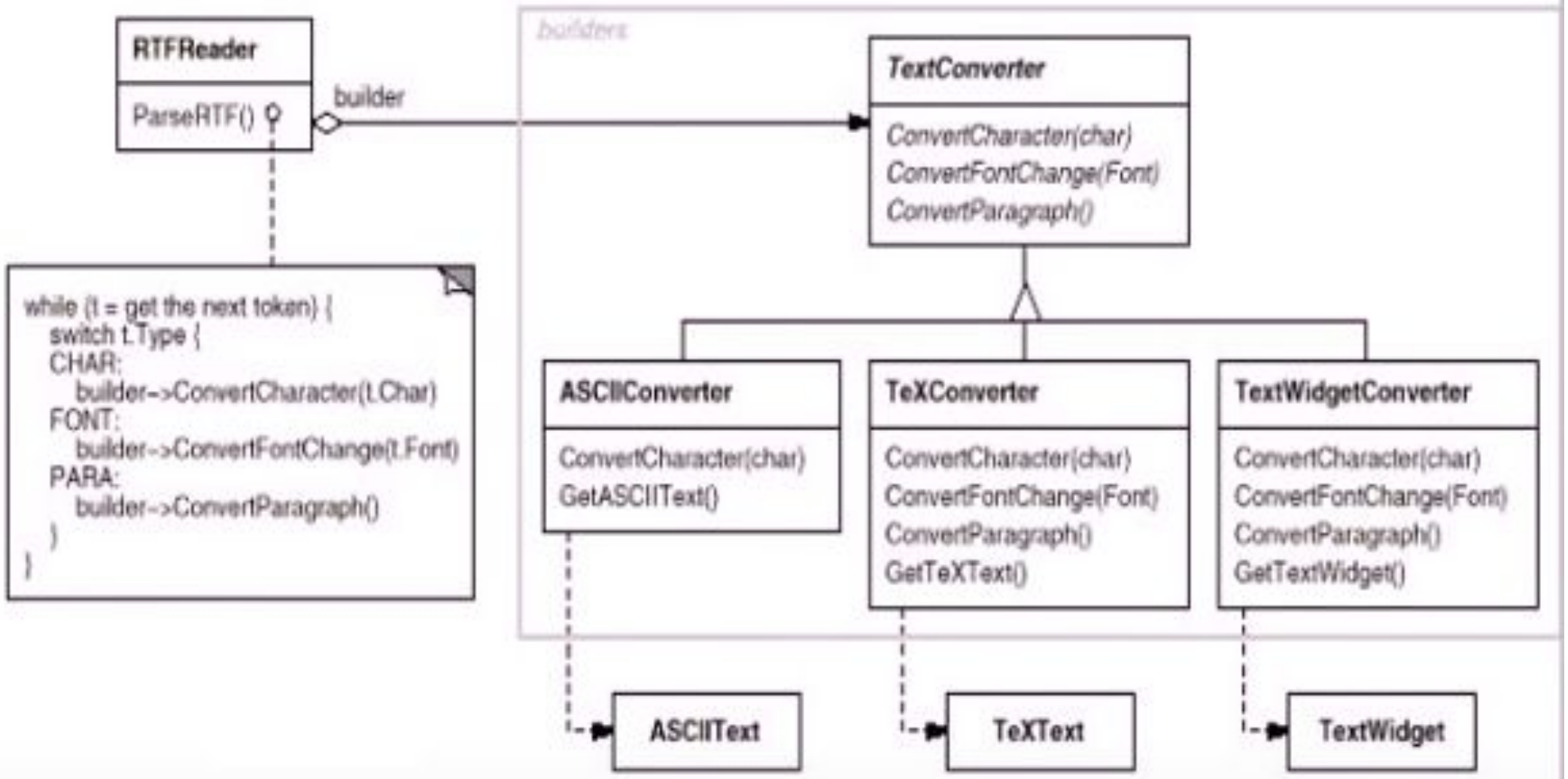
Зачем


French Translator
Use
Reader (Director)
Convertthis to French
Translator (Builder)
Builders
French Translator
German Translator
Sanskrit Translator

Мотивация


Customer
(Client)
Cashier
(Director)
Restaurant Crew
(Builder)
Order Kid's Meal
Build
Build
Build
Build
Get Meal

Мотивация


RTFReader
ParseRTF()
builder
builders
TextConverter
ConvertCharacter(char)
ConvertFontChange(Font)
ConvertParagraph()
while (t = get the next token) {
switch t.Type {
CHAR:
builder->ConvertCharacter(t.Char)
FONT:
builder->ConvertFontChange(t.Font)
PARA:
builder->ConvertParagraph()
}
}
ASCIIConverter
ConvertCharacter(char)
GetASCIIText()
TeXConverter
ConvertCharacter(char)
ConvertFontChange(Font)
ConvertParagraph()
GetTeXText()
TextWidgetConverter
ConvertCharacter(char)
ConvertFontChange(Font)
ConvertParagraph()
GetTextWidget()
ASCIIText
TeXText
TextWidget

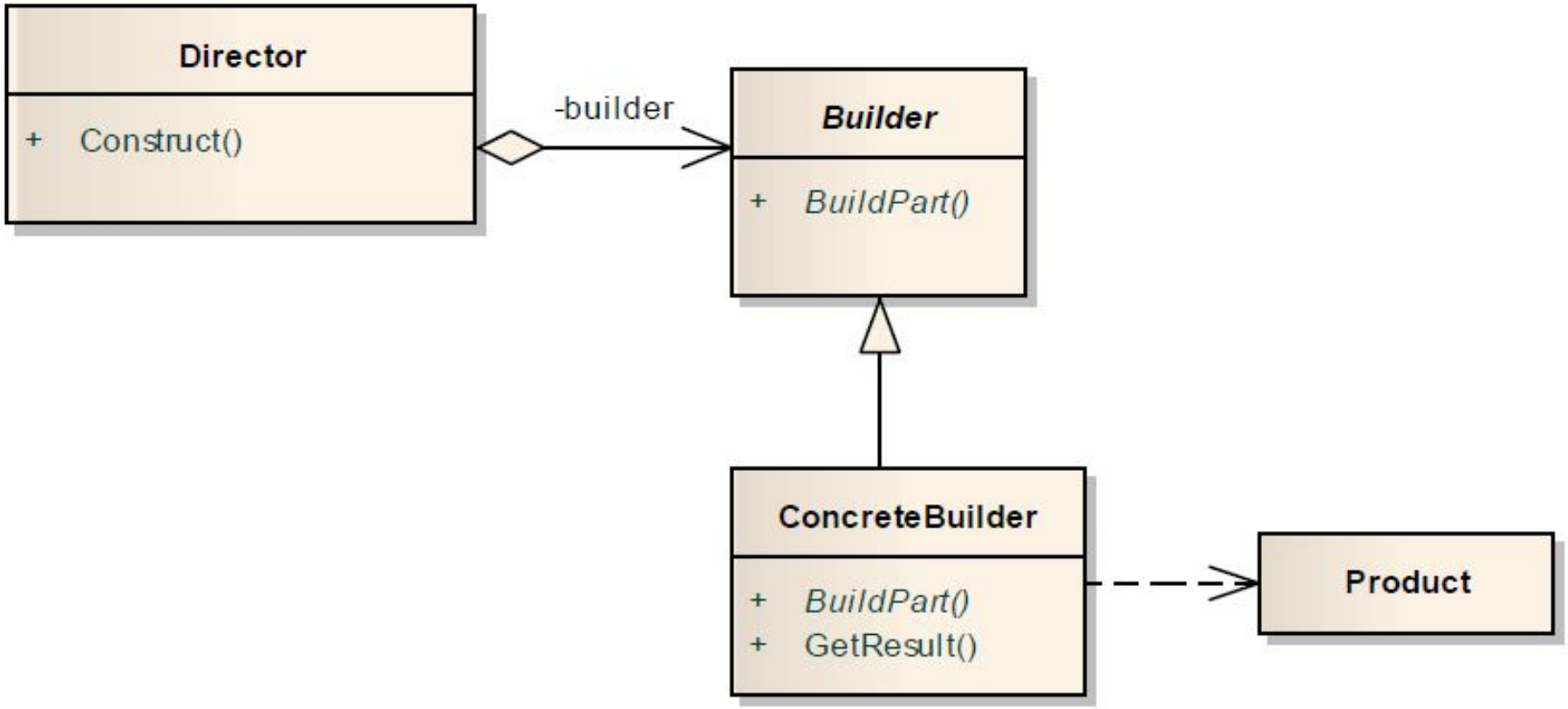
Применимость

Используйте паттерн строитель, когда:

1. алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой;

2. процесс конструирования должен обеспечивать различные представления конструируемого объекта.

Структура паттерна Builder



Участники

1. **Builder**(TextConverter) ; **строитель**: задает абстрактный интерфейс для создания частей объектаProduct;
2. **ConcreteBuilder**(ASCIIConverter,TeXConverter,TextWidgetConverter); **конкретный строитель**:- конструирует и собирает вместе части продукта посредством реализации интерфейса Builder;; определяет создаваемое представление и следит за ним; предоставляет интерфейс для доступа к продукту (например,GetASCIIText,GetTextWidget);
3. **Director** (RTFReader) ; **распорядитель**: конструирует объект, пользуясь интерфейсомBuilder;
4. **Product** (ASCIIText,TeXText,TextWidget) ; **продукт**:. - представляет сложный конструируемый объект.ConcreteBuilder -*строит внутреннее представление продукта* и определяет процесс его
   сборки;
   включает классы, которые определяют составные части, в том числе интерфейсы для сборки конечного результата из частей

Отношения

1. клиент создает объект-распорядитель Director и конфигурирует его нужным объектом-строителем Builder;

2. распорядитель уведомляет строителя о том, что нужно построить очередную часть продукта;

3. строитель обрабатывает запросы распорядителя и добавляет новые части к продукту;

4. клиент забирает продукт у строителя.

Отношения между участниками

- Клиент конфигурирует распорядителя (Director) экземпляром конкретного строителя.
- Распорядитель вызывает методы строителя для конструирования частей продукта.
- Конкретный строитель создает продукт и следит за его конструированием.
- Конкретный строитель представляет интерфейс для доступа к продукту.

Результаты

1. позволяет изменять внутреннее представление продукта
2. изолирует код, реализующий конструирование и представление
3. дает более тонкий контроль над процессом конструирования

Реализация

1. интерфейс сборки и конструирования - **должен быть достаточно общим**

2. нет абстрактного класса для продуктов (**продукты сильно разнятся** - потому нет смысла в общем интерфейсе)

3. пустые методы класса Builder по умолчанию (а не виртульные) - специально для того чтобы их *не обязательно было реализовывать в каждом конкретном строителе*

.

Пример кода на C++

```
// Product
class Pizza
{
private:
    std::string dough;
    std::string sauce;
    std::string topping;

public:
    Pizza() { }
    ~Pizza() { }
    void SetDough(const std::string& d) { dough = d; }
    void SetSauce(const std::string& s) { sauce = s; }
    void SetTopping(const std::string& t) { topping = t; }
    void ShowPizza()
    {
        std::cout << " Yummy !!!" << std::endl
        << "Pizza with Dough as " << dough
        << ", Sauce as " << sauce
        << " and Topping as " << topping
        << " !!! " << std::endl;
    }
 };
```

```
// Director
class Waiter
{
private:
   PizzaBuilder* pizzaBuilder;
public:
   Waiter() : pizzaBuilder(NULL) {}
   ~Waiter() { }
    void SetPizzaBuilder(PizzaBuilder* b) { pizzaBuilder = b; }
   std::shared_ptr<Pizza> GetPizza() { return pizzaBuilder->GetPizza(); }
   void ConstructPizza()
   {
      pizzaBuilder->createNewPizzaProduct();
      pizzaBuilder->buildDough();
      pizzaBuilder->buildSauce();
      pizzaBuilder->buildTopping();
   }
};                          // Клиент заказывает две пиццы.
```

```
// Abstract Builder
class PizzaBuilder
{
protected:
    std::shared_ptr<Pizza> pizza;
public:
    PizzaBuilder() {}
    virtual ~PizzaBuilder() {}
    std::shared_ptr<Pizza> GetPizza() { return pizza; }
     void createNewPizzaProduct() { pizza.reset (new Pizza); }
     virtual void buildDough()=0;
    virtual void buildSauce()=0;
    virtual void buildTopping()=0;
 };
```

----> …

```
class HawaiianPizzaBuilder : public PizzaBuilder
{
public:
    HawaiianPizzaBuilder() : PizzaBuilder() {}
    ~HawaiianPizzaBuilder(){}
     void buildDough() { pizza->SetDough("cross"); }
    void buildSauce() { pizza->SetSauce("mild"); }
    void buildTopping() { pizza->SetTopping("ham and pineapple"); }
};
```

```
// ConcreteBuilder
class SpicyPizzaBuilder : public PizzaBuilder
{
public:
    SpicyPizzaBuilder() : PizzaBuilder() {}
    ~SpicyPizzaBuilder() {}

    void buildDough() { pizza->SetDough("pan baked"); }
    void buildSauce() { pizza->SetSauce("hot"); }
    void buildTopping() { pizza->SetTopping("pepperoni and salami"); }
};
```

C:\Windows\system32\cmd.exe

```
 Yummy !!!
Pizza with Dough as cross, Sauce as mild and Topping as ham and pineapple !!!
 Yummy !!!
Pizza with Dough as pan baked, Sauce as hot and Topping as pepperoni and salami
!!!
Для продолжения нажмите любую клавишу . . .
```

Два секрета

1. Знание о конкретных классах
2. Знание о их стыковке

Результаты использования паттерна

- Есть возможность изменять внутреннею структуру создаваемого продукта (или создать новый продукт).
  - продукт конструируется через абстрактный интерфейс класса Builder, для добавления нового продукта достаточно определить новый вид строителя (т.е. реализовать новый подкласс класса Builder).
- Повышение модульности за счет разделения распорядителя и строителя.
  - Каждый строитель имеет весь необходимый код для пошагового построения продукта.
  - Поэтому он может использоваться разными распорядителями для построения вариантов продукта из одних и тех же частей.
- Пошаговое построение продукта позволяет обеспечить более пристальный контроль над процессом конструирования
  - в отличие от других порождающих паттернов, которые создают продукт мгновенно.

Лабораторная работа №1 *(дедлайн - 26.02)*

Порядок выполнения работы:

1. С использованием одного из языков программирования (C++) реализовать шаблоны проектирования

а) ***Шаблон “Абстрактная фабрика”.*** *Проект “Заводы по производству автомобилей”. В проекте должно быть реализована возможность создавать автомобили различных типов на разных заводах.*

б) *Хлеб может иметь различную комбинацию компонентов: ржаной и пшеничной муки, соли, пищевых добавок. И нам надо обеспечить выпечку разных сортов хлеба. Для разных сортов хлеба может варьироваться конкретный набор компонентов, не все компоненты могут использоваться. И для этой задачи применим паттерн* ***Builder.***

2. Разработать UML модель