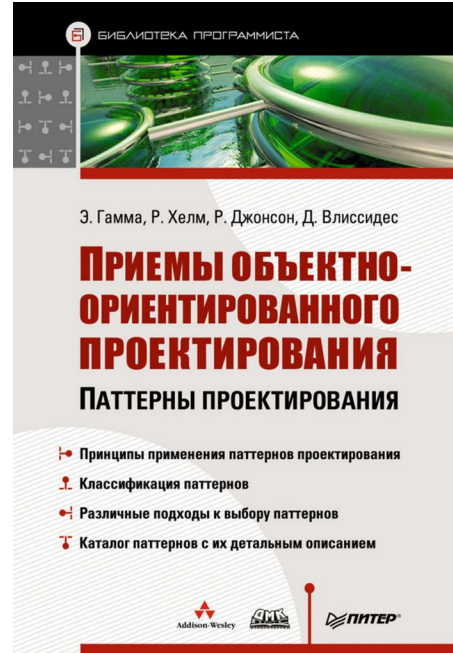


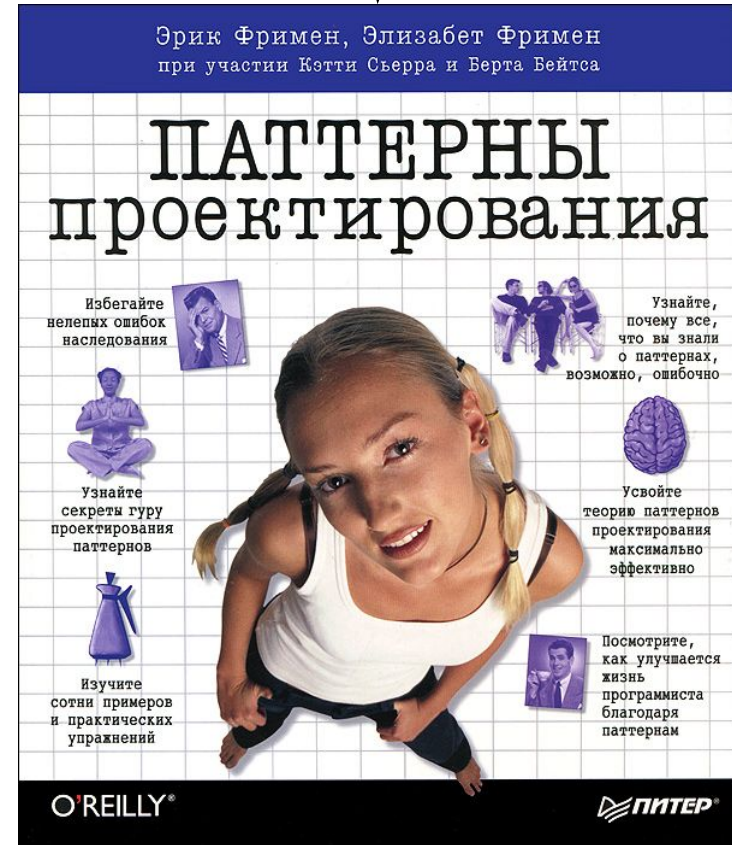
Design Patterns

- Design Pattern Definition. Pattern Classification
- Common GoF Patterns
 - Strategy
 - Decorator
 - Iterator. External and Internal Iteration
 - Composite (Tree)
 - Observer. Events and Event Objects
 - Proxy. Remote Stubs. Decorator vs Proxy
 - Facade
 - Creational Patterns: Singleton, Abstract Factory, Builder
- Some Pattern Pitfalls

Best Intro into Patterns



The Classic GoF Book
Same book, different cover



Design Pattern is...

https://en.wikipedia.org/wiki/Software_design_pattern

- **General, Reusable** solution to a **commonly occurring** problem
 - within a given **Context**
 - in **Software Design**
 - e.g., Christopher Alexander described patterns in **architecture**
- **Description** or **Template** that can be used in **many different situations**
 - Shows relationships and interactions between classes or objects **in general**
 - Facilitates **Common Language** between developers
- Formalized **best practice**
- Patterns are Programming Paradigm-Dependent!
 - Some OO Patterns are Functional Programming in disguise
 - E.g. <http://www.norvig.com/design-patterns/design-patterns.pdf>

Pattern Classification



Creational

- Abstract Factory
- Factory Method
- Builder
- Singleton
- Prototype
- ...

Structural

- Decorator
- Proxy
- Facade
- Adapter
- Flyweight
- ...

Behavioral

- Strategy
- Iterator
- Observer
- State
- Template Method
- Visitor
- ...

GoF Book: Cake is a Lie!

The Classic *Design Patterns* book is mostly **dense text**

- Pattern Rationale
- Source code in C++
- And some diagrams
 - In **OMT**, not UML

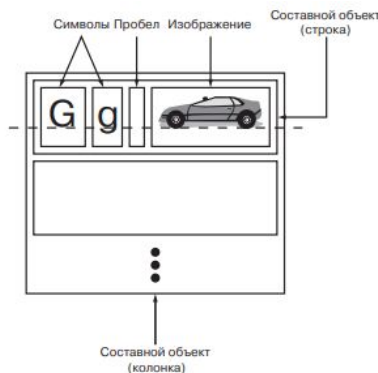
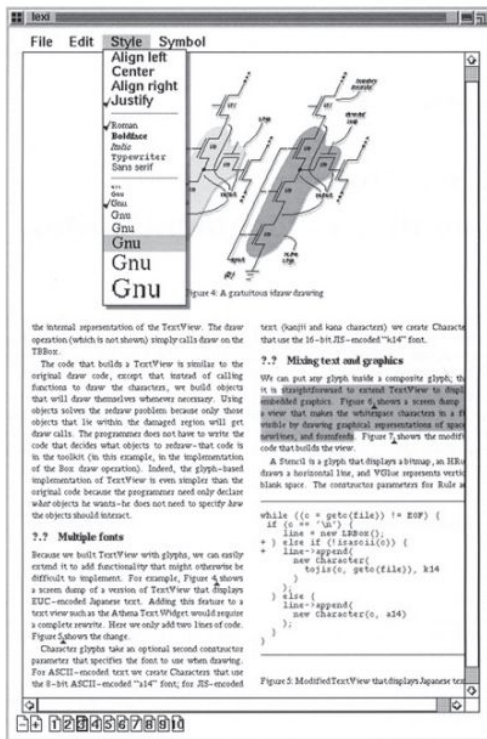


Рис. 2.2. Рекурсивная композиция текста и графики



Рис. 2.3. Структура объекта для рекурсивной композиции текста и графики

Pattern Description in GoF Book

- **Pattern Name, Classification, AKA**
- **Intent:** Goals + Reason to Use
- **Motivation:** Example Problem + Context
- **Applicability:** All Suitable Contexts
- **Structure:** Classes & Interactions
- **Participants:** Classes + Roles
- **Collaboration:** Class Interactions
- **Consequences:**
 - Results, Side Effects
 - Tradeoffs
- **Related Patterns**

2022-04-24

112

Порождающие паттерны

Известен также под именем Virtual Constructor (виртуальный конструктор).

Мотивация

Каркасы пользуются абстрактными классами для определения и поддержания отношений между объектами. Кроме того, каркас часто отвечает за создание самих объектов.

Рассмотрим каркас для приложений, способных представлять пользователю сразу несколько документов. Две основных абстракции в таком каркасе – это классы `Application` и `Document`. Оба класса абстрактные, поэтому клиенты должны порождать от них подклассы для создания специфичных для приложения реализаций. Например, чтобы создать приложение для рисования, мы определяем классы `DrawingApplication` и `DrawingDocument`. Класс `Application` отвечает за управление документами и создает их по мере необходимости, допустим, когда пользователь выбирает из меню пункт `Open` (открыть) или `New` (создать).

Поскольку решение о том, какой подкласс класса `Document` инстанцировать, зависит от приложения, то `Application` не может «предсказать», что именно понадобится. Этому классу известно лишь, когда нужно инстанцировать новый документ, а не какой документ создать. Возникает дилемма: каркас должен инстанцировать классы, но «знает» он лишь об абстрактных классах, которые инстанцировать нельзя.

Решение предлагает паттерн фабричный метод. В нем инкапсулируется информация о том, какой подкласс класса `Document` создать, и это знание выводит за пределы каркаса.

Подклассы класса `Application` переопределяют абстрактную операцию `CreateDocument` таким образом, чтобы она возвращала подходящий подкласс класса `Document`. Как только подкласс `Application` инстанцирован, он может инстанцировать специфические для приложения документы, ничего не зная об их классах. Операцию `CreateDocument` мы называем **фабричным методом**, поскольку она отвечает за «изготовление» объекта.

113

Паттерн Factory Method

ПРИМЕНИМОСТЬ

Используйте паттерн фабричный метод, когда:

- классу заранее неизвестно, объекты каких классов ему нужно создавать;
- класс спроектирован так, чтобы объекты, которые он создает, специфицировались подклассами;
- класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и вы планируете локализовать знание о том, какой класс принимает эти обязанности на себя.

Структура

Участники

- **Product** (`Document`) – продукт:
 - определяет интерфейс объектов, создаваемых фабричным методом;
- **ConcreteProduct** (`MyDocument`) – конкретный продукт:
 - реализует интерфейс `Product`;
- **Creator** (`Application`) – создатель:
 - объявляет фабричный метод, возвращающий объект типа `Product`. `Creator` может также определять реализацию по умолчанию фабричного метода, который возвращает объект `ConcreteProduct`;
 - может вызывать фабричный метод для создания объекта `Product`;
- **ConcreteCreator** (`MyApplication`) – конкретный создатель:
 - замещает фабричный метод, возвращающий объект `ConcreteProduct`.

Отношения

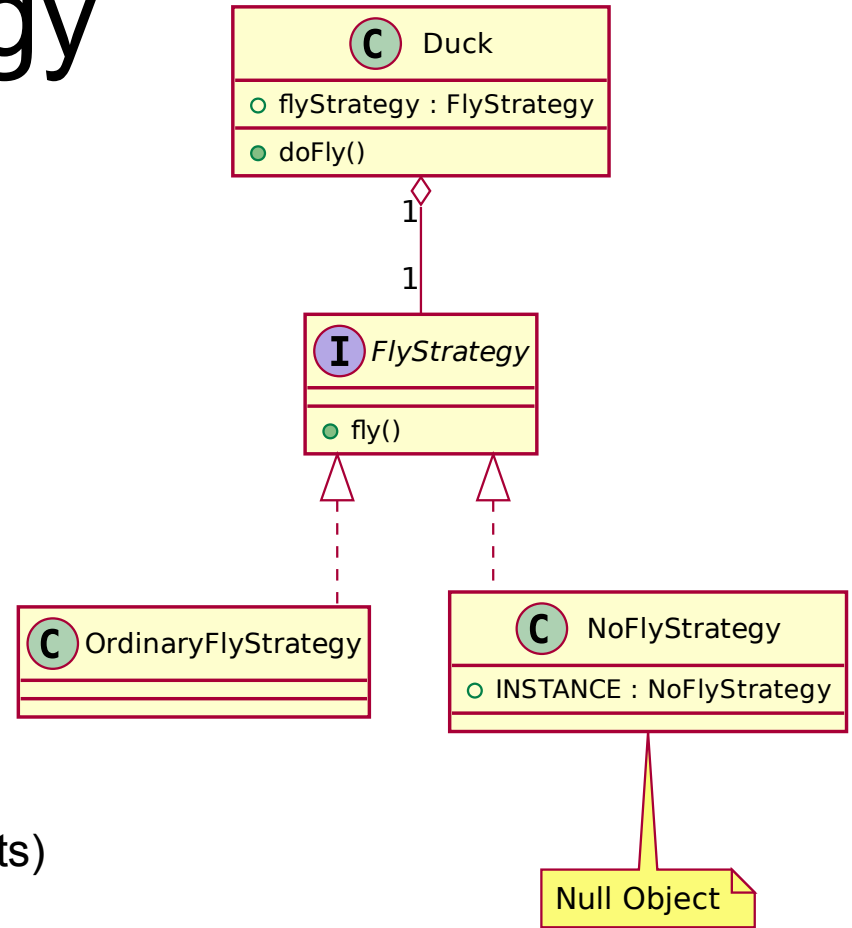
Создатель «полагается» на свои подклассы в определении фабричного метода, который будет возвращать экземпляр подходящего конкретного продукта.

Результаты

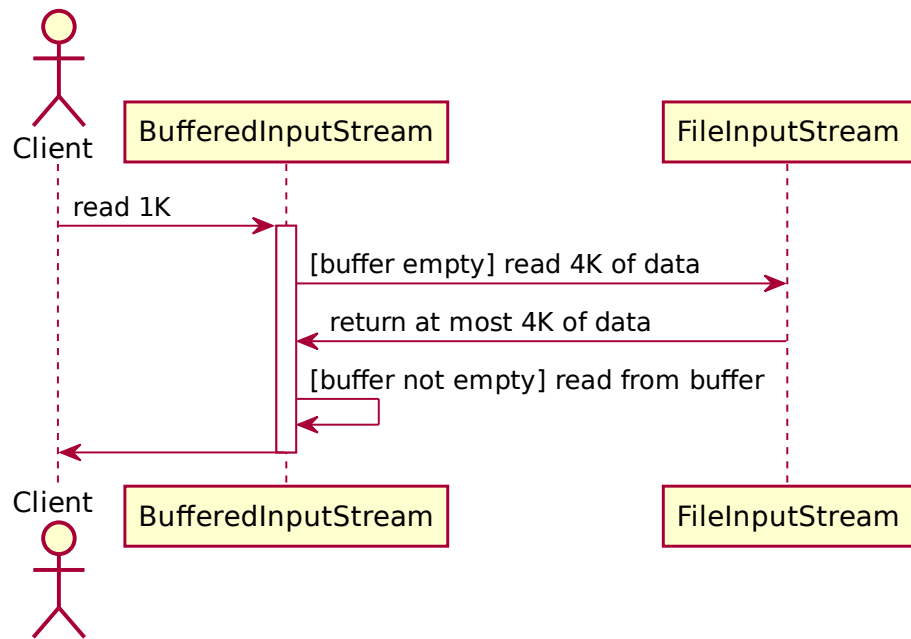
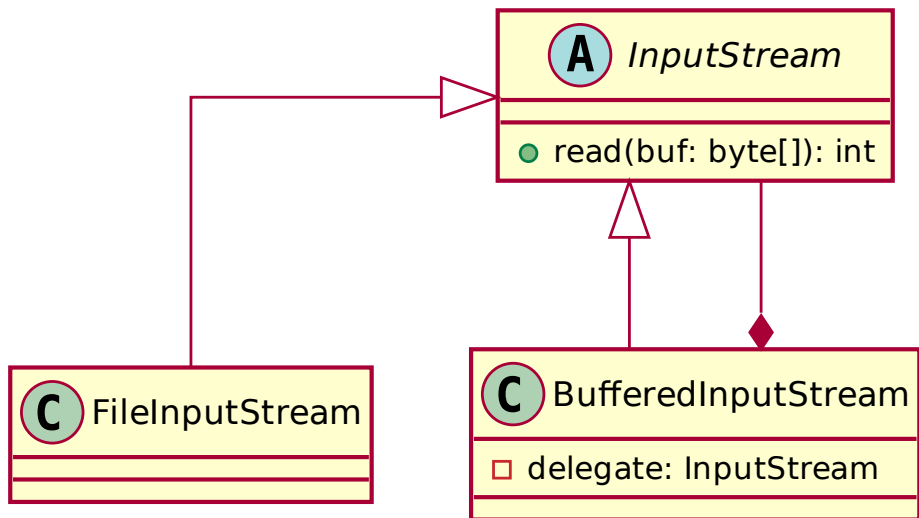
Фабричные методы избавляют проектировщика от необходимости встраивать в код зависящие от приложения классы. Код имеет дело только с интерфейсом класса `Product`, поэтому он может работать с любыми определенными пользователями классами конкретных продуктов.

Strategy

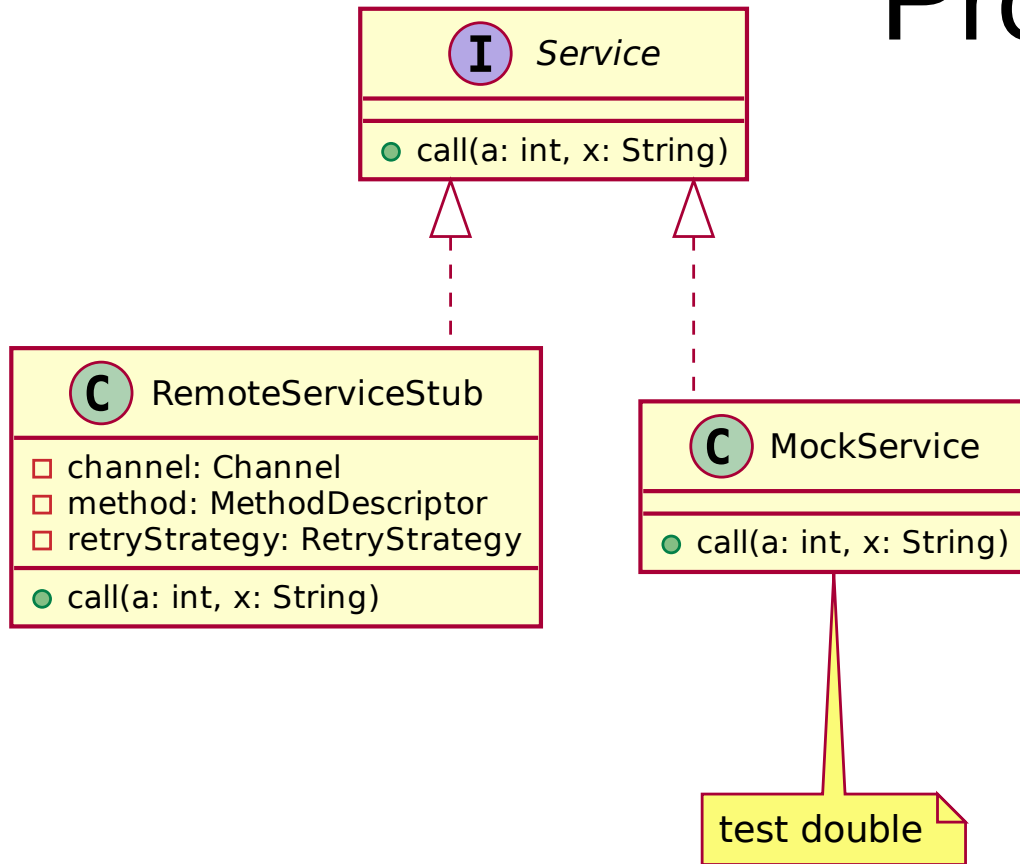
- Duck.fly(). RubberDuck extends Duck. **Oops!**
- **Solution:**
 - FlyStrategy.fly()
 - Duck <<use>> FlyStrategy.fly()
 - RubberDuck = (Duck, NoFlyStrategy)
 - Ordinary Duck = (Duck, OrdinaryFlyStrategy)
- Strategy offers **Pluggable Behavior**
 - e.g. Retry Strategy
- Strategy User is implementation-independent
 - Just requires that impl has the specified *interface* (=public methods, pre-and postconditions, invariants)



Decorator



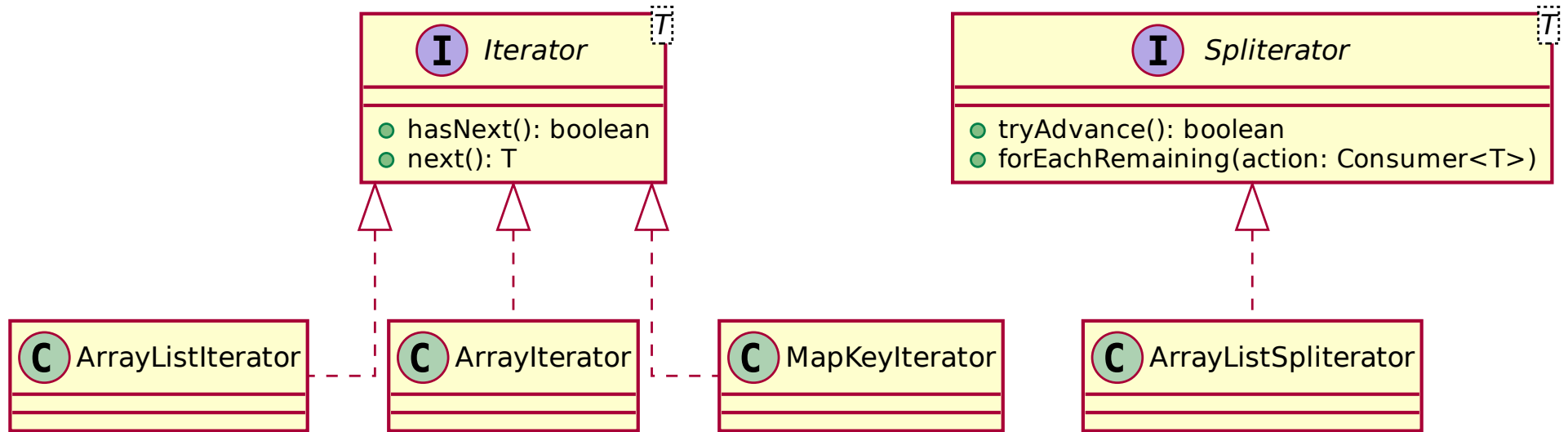
Proxy



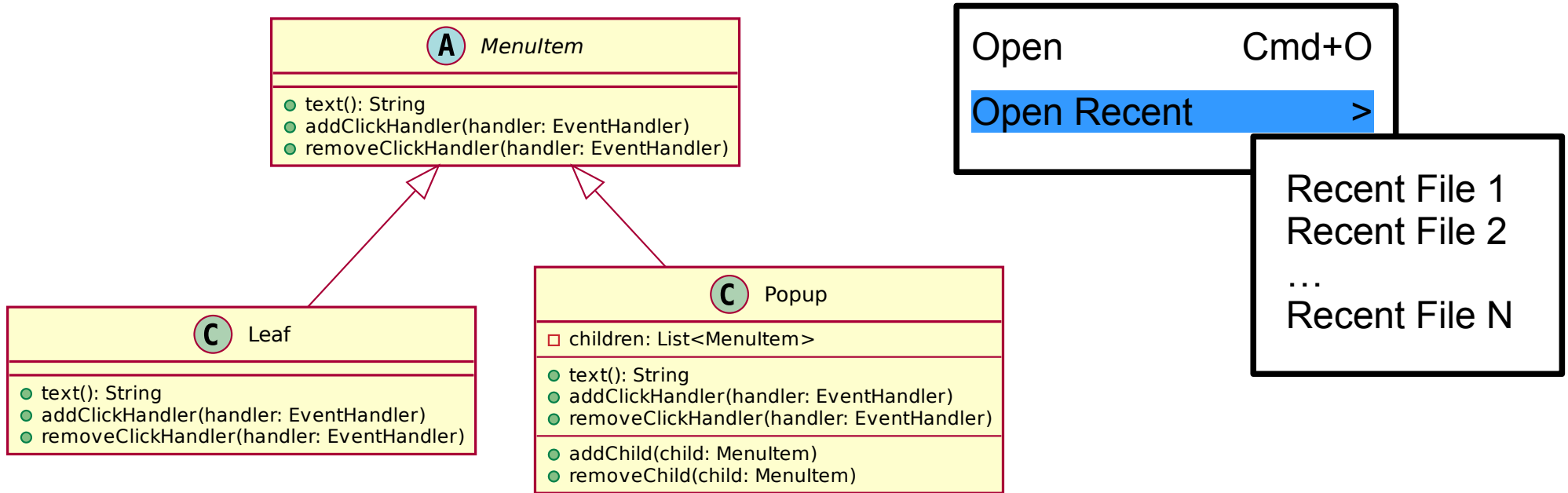
Proxy vs Decorator:

- Decorators *can* have **similar implementations** as Proxies but serve a **different purpose**
 - E.g.* protection proxy, logging proxy
- Decorator **adds one or more responsibilities** to an object
- Proxy **controls access** to an object
- Unlike a Decorator:
 - RPC Proxy (e.g. `RemoteServiceStub`) **will not contain a direct reference** to the real subject
 - Virtual Proxy (e.g. reading data from file on demand) **will not contain a direct reference to the real data** until called

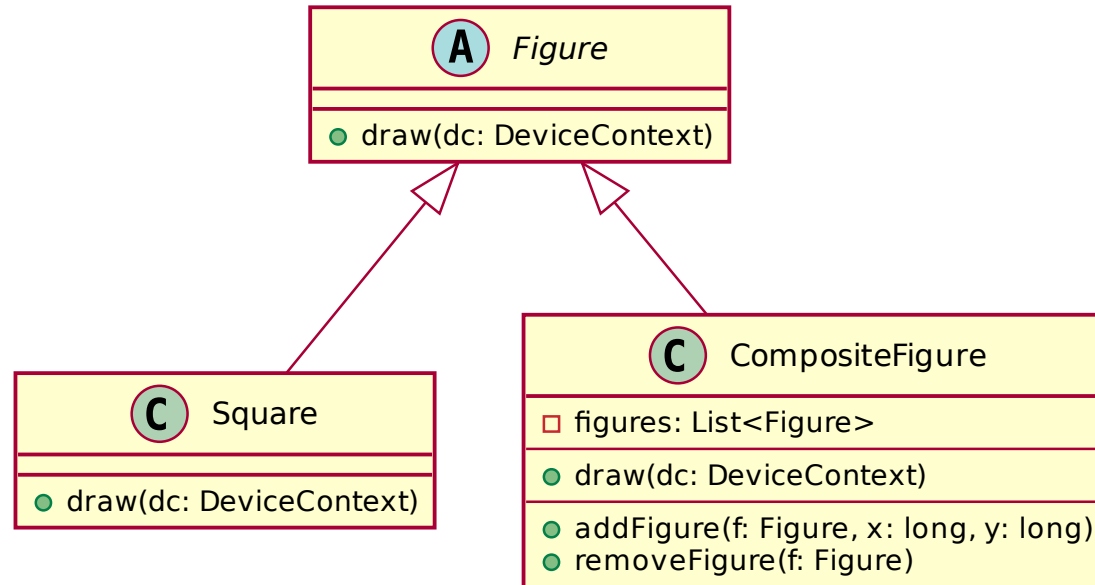
Iterator



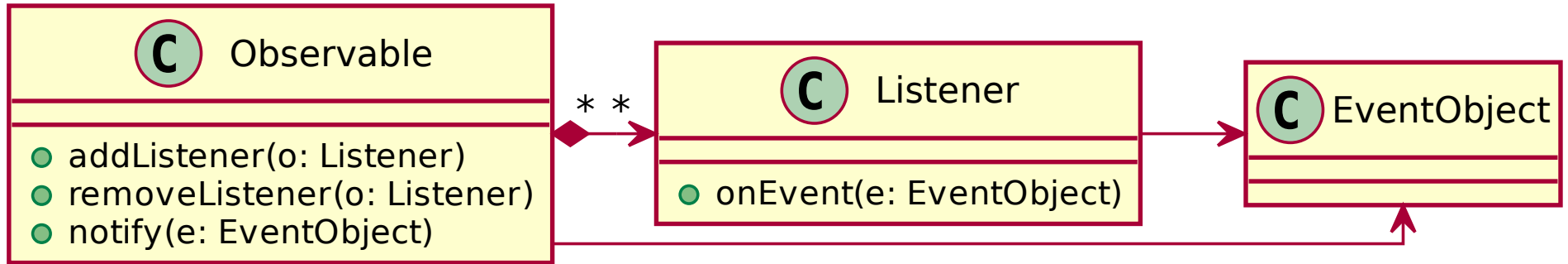
Composite (Tree)



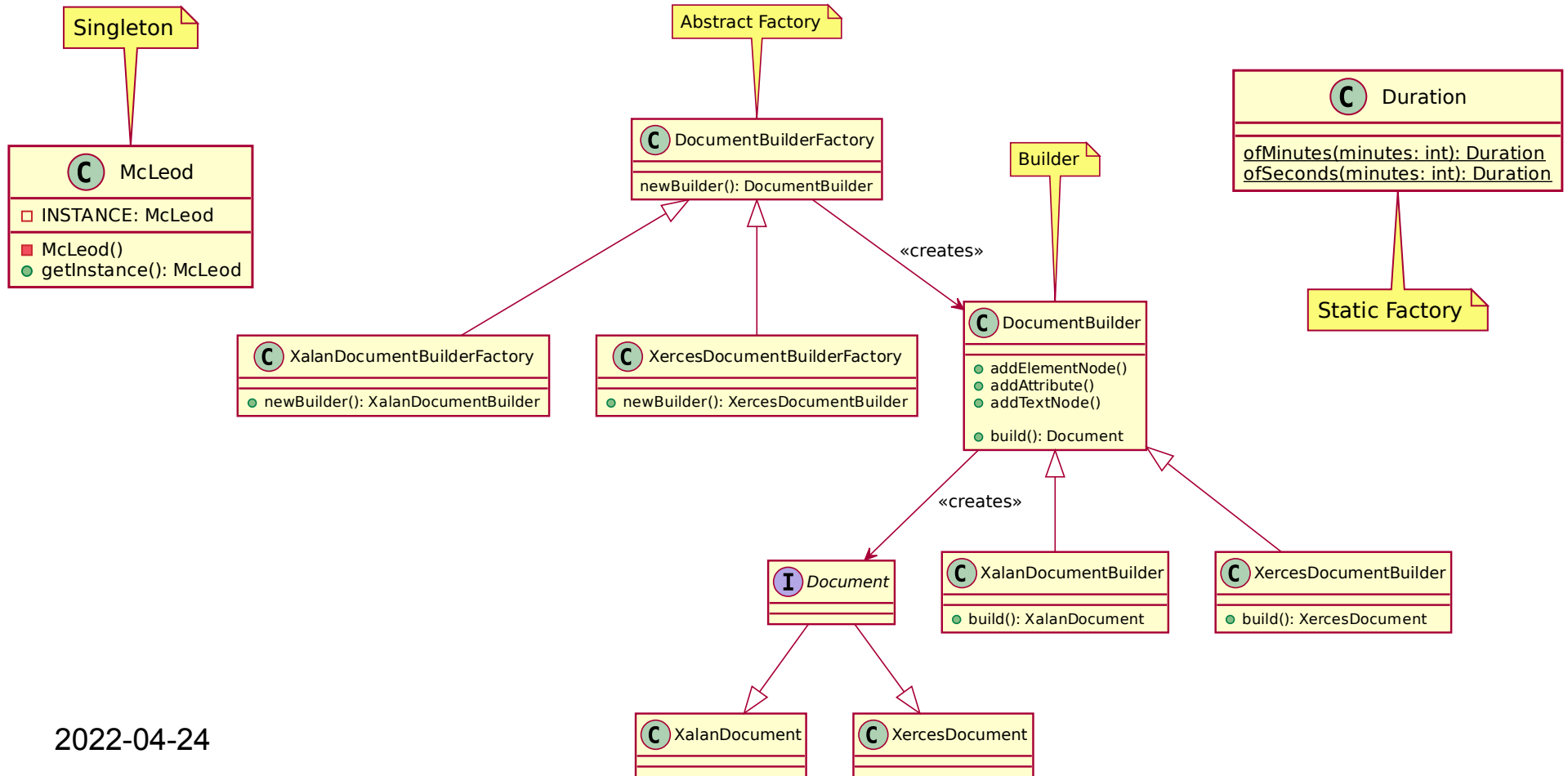
Composite (Tree)



Observer



Common Creational Patterns



Some Pattern Pitfalls

- I hate `switch` and `Map`, let's use **polymorphism**
 - <https://csis.pace.edu/~bergin/patterns/ppoop.html>
- We Need Flexibility Everywhere!
 - ...and then it turns out the interface always has 1 impl
 - System Architecture is what **cannot be changed** without **completely destroying** the System

Recommended Reading (List)

- *Head First Patterns* by Eric & Elizabeth Freeman
 - <https://www.ozon.ru/product/head-first-patterny-proektirovaniya-obnovlennoe-yubileynoe-izdanie-frimen-erik-robson-elizabet-211433204>
- *Design Patterns: Elements of Reusable Object-Oriented Software* by E. Gamma, R. Helm, R. Johnson, D. Vlissides
 - <https://www.ozon.ru/product/priemy-obektno-orientirovannogo-proektirovaniya-patterny-proektirovaniya-135466040/>