ADDITIONAL MATERIAL FOR INFSEN01-2

DR ANNA V. KONONOVA

LECTURE 6. DECORATOR PATTERN. JUNE 6, 2017

- Reminder: **Read all official slides as well** (your exam is prepared based on them)
- Photo opportunity!
- Last week, we have revisited the "Heating element" example, studied abstract factory and factory method patterns, worked on the "Bad Santa" example
- **Today** we will: quickly revise last lesson's example, look at the decorator patterns and work on a new example problem.

EXAMPLE: SANTA'S WORKSHOP

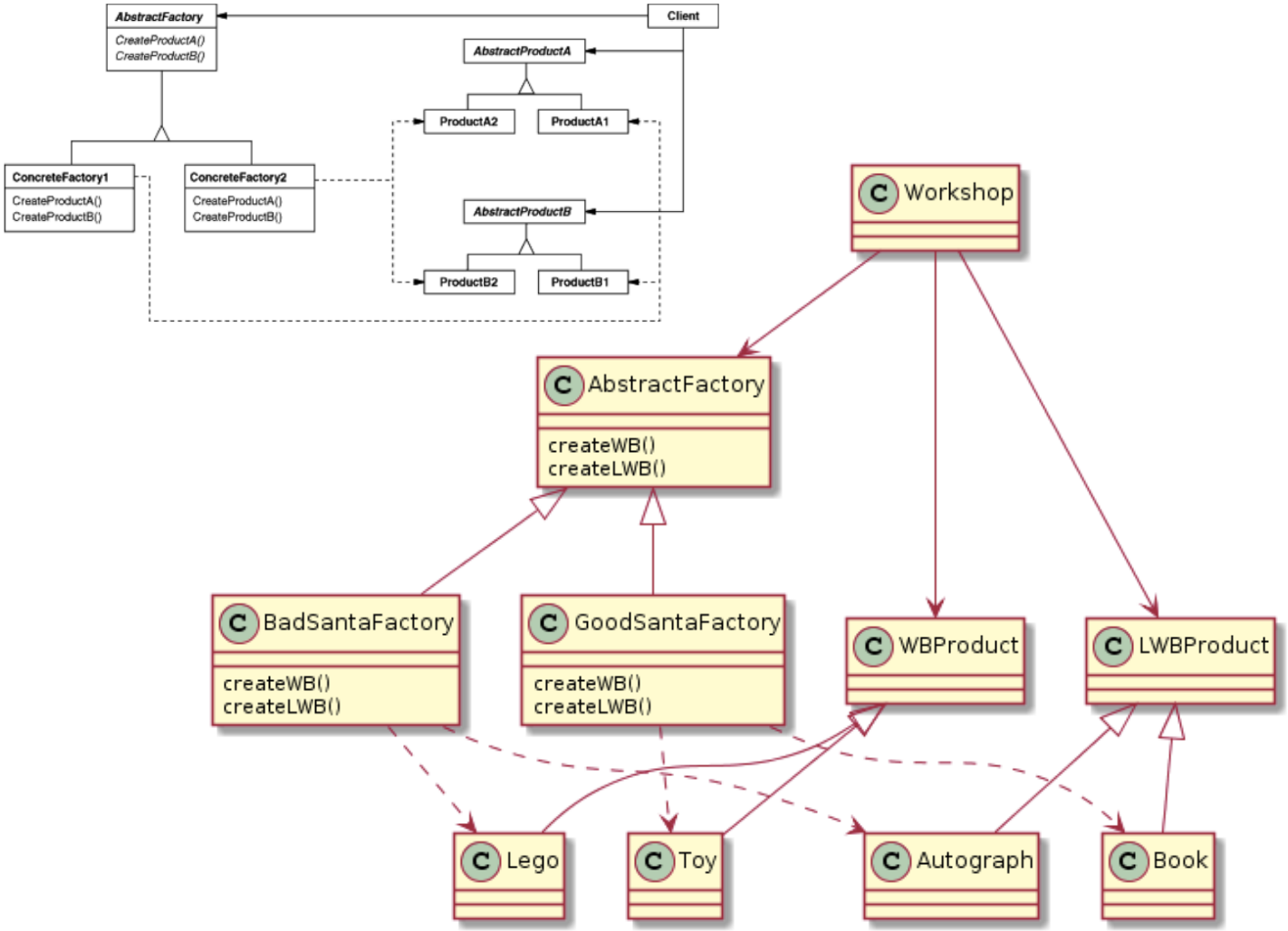
- "**Santa's Workshop**": due to a rapid growth of the Earth's population, Santa Claus had to work multiple shifts. To reduce his workload, an extra Santa has been hired. Unfortunately only Bad Santa was available. Both Santas prepare two kinds of presents: one for well-behaved kids and one for the less-well-behaved kids. However, clearly their presents differ.. a little
- *Good Santa* chooses to give every *well-behaved* kid a hand-made toy of their choice. *Less-well-behaved* kids receive a freshly printed "How to be a good child" book.
- *Bad Santa* prepares presents for *well-behave*d kids packing random parts of various Lego sets. Meanwhile the *less-well-behaved* kids receive his real autograph
- Both Santas work simultaneously and choice of a Santa is done runtime based on each Santa's availability

➢ design and draw the class diagram to implement the application running the joint Santa's workshop

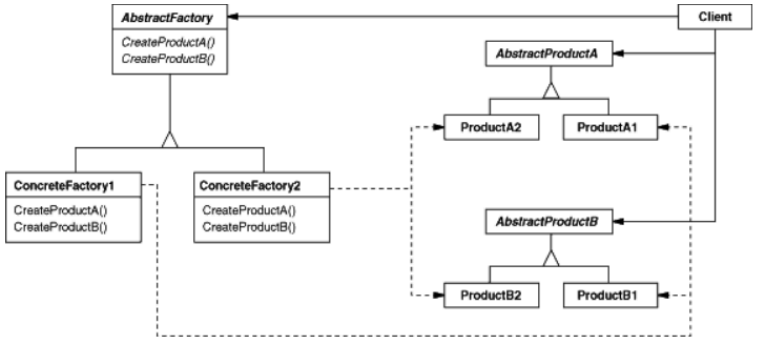
➢ run the workshop with requests for 5 kids: kids 1,2, 4 are well-behaved and kids 3, 5 are less-well-behaved. You can choose which Santa is available for each kid

* several designs are possible

EXAMPLE: SANTA'S WORKSHOP



EXAMPLE: SANTA'S WORKSHOP

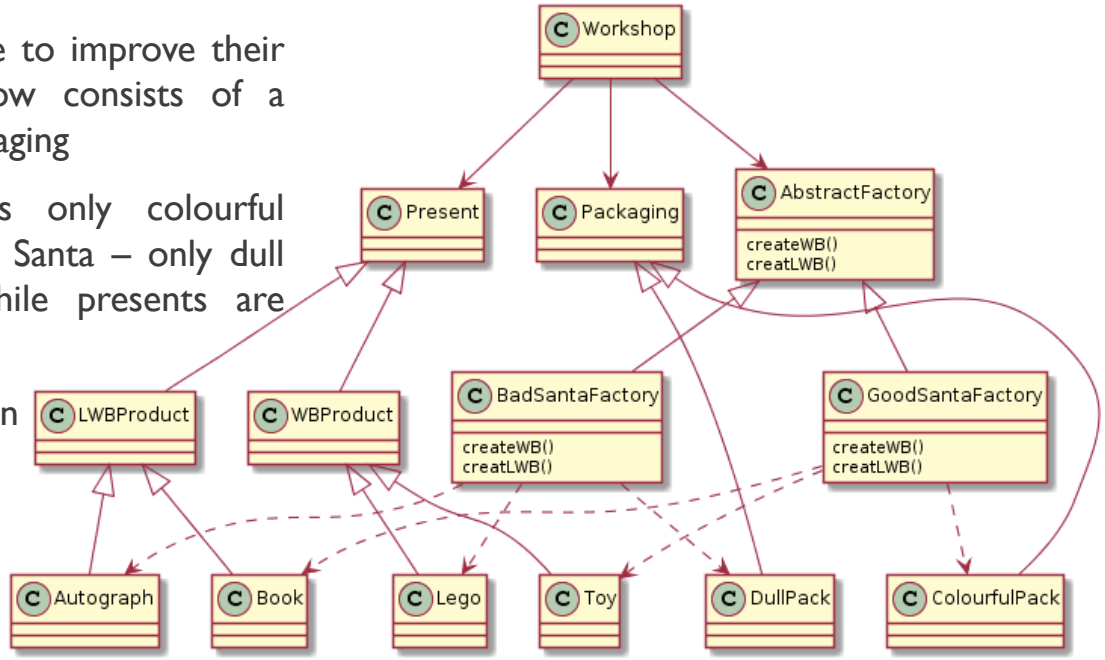


Both Santas decide to improve their gifts: each gift now consists of a present and a packaging

Good Santa uses only colourful packaging and Bad Santa – only dull packaging. Meanwhile presents are chosen as before

➢redraw the design

* several designs are possible



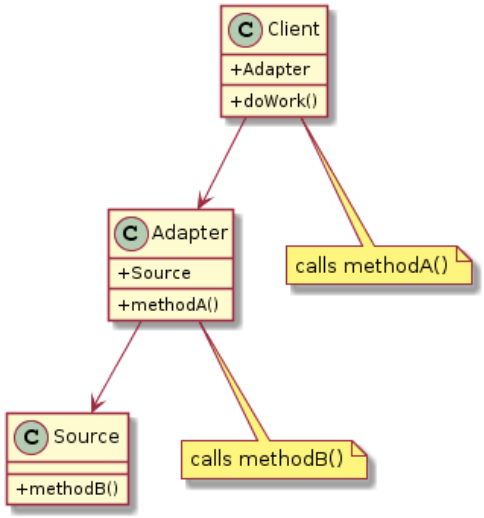
DECORATOR: INTENT, MOTIVATION AND APPLICABILITY

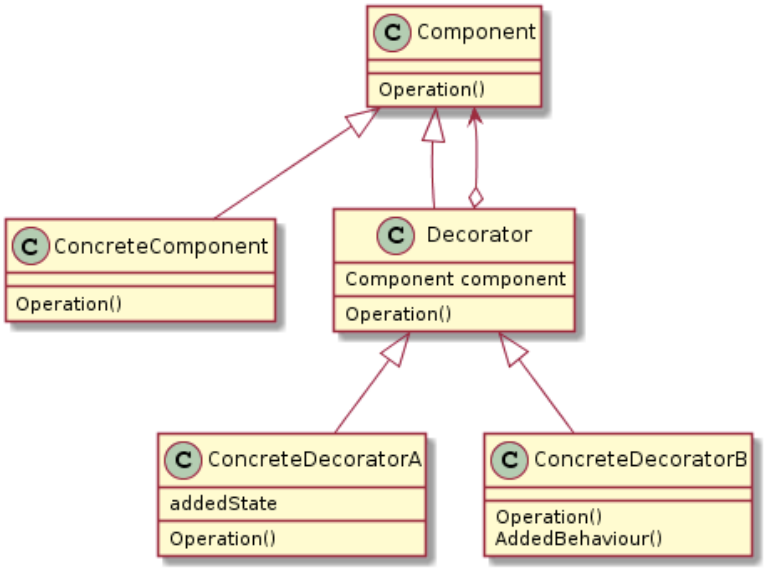
- **Adapter** pattern is used to convert the interface of a class into another interface that is expected by a client. ..Adapter may not change or add behaviours. *Remember?*
- Need to add behaviours? ➢ inheritance seems logical choice
- However it is done statically and breaks one of the OO principles (*which?*)
- Favour composition over inheritance (*remember?*) ➢ **decorator** pattern
- **Intent**: attach additional responsibilities to an object dynamically
- **Motivation**: decorator conforms to the interface of the components it decorates so that its presence is transparent to the component's clients. Transparency allows to nest decorations recursively, even repetitevely (➢ unlimited number of added responsibilities). Clients do not depend on decorations.
- **Applicability**:

- To add responsibilities to individual objects dynamically and transparently
- For responsibilities that can be withdrawn
- When extension by subclassing is impractical (danger of the "explosion" of subclasses or class definition not being available)

DECORATOR: STRUCTURE AND PARTICIPANTS

- **Component** – defines the interface for objects that can have responsibilities added to them dynamically
- **ConcreteComponent** – defines an object to which additional responsibilities can be attached
- **Decorator** – maintains a reference to a Component object and defines an interface that conforms to Component's interface; forwards requests to its Component object, may perform additional operations before and after the forwarding
- **ConcreteDecorator** – adds responsibilities to the component





DECORATOR: CONSEQUENCES AND IMPLEMENTATION

Consequences:

- (+) *More flexibility than static inheritance* (simpler class structure, recursive decorating, easy extension for new responsibilities)
- (+) *Avoids feature-laden classes high up in the hierarchy* (adding decorations in a pay-as-you-go fashion, composing functionality on the fly)
- (-) *A decorator and its component are not identical.* Decorator is transparent enclosure of the component. From object point of view a decorated component ! ≡ component itself ➢ don't rely on object identity when using decorators
- (-) *Lots of little objects*

Implementation:

- *Interface conformance.* A decorator object must confirm to the interface of Component ➢ ConcreteDecorators must inherit from common class
- *Omitting the abstract Decorator class.* If only one responsibility Decorator and ConcreteDecorator can be merged in one class
- *Keeping Component class lightweight.* To ensure conforming interface components and decorators must descend from a common Component class ➢it is used for defining an interface, not storing data ➢ definition of data representation should be deferred to subclasses
- *Changing the skin of an object versus changing its guts.* Decorator is a skin changing behaviour. Alternatively, object can change its guts (Strategy pattern) ➢own choice

Decorator ➢ component knows nothing about decorators. Strategy ➢ component knows about possible extensions. New extensions with Strategy ➢ modifying components. Strategy ➢ own specialised interface is possible whereas Decorator must conform …

DECORATOR: OFFICIAL SLIDES

- Given a polymorphic type *I* (to instantiate) and a series of concrete implementations of *I*: C_1, …, C_m, a decorator *D* is an entity that implements *I* and references and instance of *I* and a series of concrete decorators CD_1, …., CD_n where each extends *D*
- Generic decorators can be built where genericity comes from lambdas which act like holes in their behaviours. Concrete decorators can be defined by specifying the underlying iterator and lambdas
- Check examples in the official slides! (pattern definition through examples)

EXAMPLE: "SUITS"

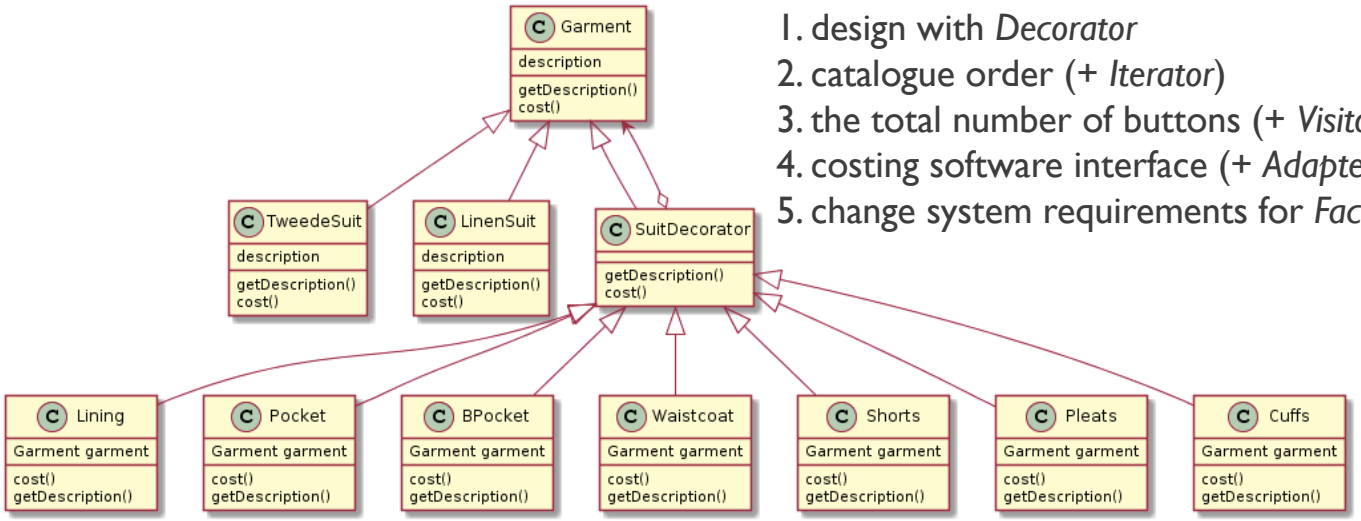
- "**Suits**": an atelier produces bespoke made-to-measure tweed or linen suits for men. The same set of body measurements is required for all suits. Suit's construction starts with a *standard suit* of the chosen fabric. When ordering, every client can fully customise all elements of the suit up intill the moment of fitting
- *Options include*: upgrading lining fabric, adding pockets on the outside of the jacket,, adding a breast pocket, adding a waistcoat, changing lower part to shorts, adding trouser pleats, adding trouser cuffs

➢ **Design classes to be used for the billing software for the atelier**

1. Use only *Decorator* pattern
2. Redesign the above for the production of the atelier's catalogue which includes all possible designs (combine *Decorator* and *Iterator*)
3. Redesign the above to calculate the total number of buttons required for the catalogue (*Visitor* pattern)
4. What if the third-party costing software requires a particular interface (*Adapter* pattern)
5. Think of a change in system requirements that can require use of some *Factory* pattern

EXAMPLE: "SUITS"

- Start with a standard suit of the chosen fabric, decorate it according to the of the customer – ConcreteDecorators update own Garment's description
- + Iterator ➤ traverse all possible decorator combinations required for the catalogue
- + Visitor ➤ add all buttons requirements for catalogue order with a ConcreteVisitor
- + Adapter ➤ adapter for costing software
- For example, consider the atelier to increase the material for suits and/or start making suits for women ➤ + some Factory pattern to create SuitMen, SuitWomen or TweedeSuit, LinenSuit, FlannelSuit, GabardineSuit, …



1. design with *Decorator*
2. catalogue order (+ *Iterator*)
3. the total number of buttons (+ *Visitor*)
4. costing software interface (+ *Adapter*)
5. change system requirements for *Factory*

CONCLUSION

- **Today** we have revised last lesson's example, studied a new design patterns and applied it to a new example problem
- **You should be nearly done with your assignment by now**

✓ Lesson 1: Introduction

✓ Lesson 2: Visitor pattern

✓ Lesson 3: Iterator pattern

✓ Lesson 4: Adapter pattern

✓ Lesson 5: Abstract factory, Factory Method

✓ Lesson 6: Decorator pattern

⏳ Lesson 7: Revision

⏳ Lesson 8: Mock exam

⏳ Lesson 7: Revision ➤ **what needs to be revised?** (June 13 2017)

- Question 1: heap, stack, declaration tasks
- Question 2: **?**
- Question 3:

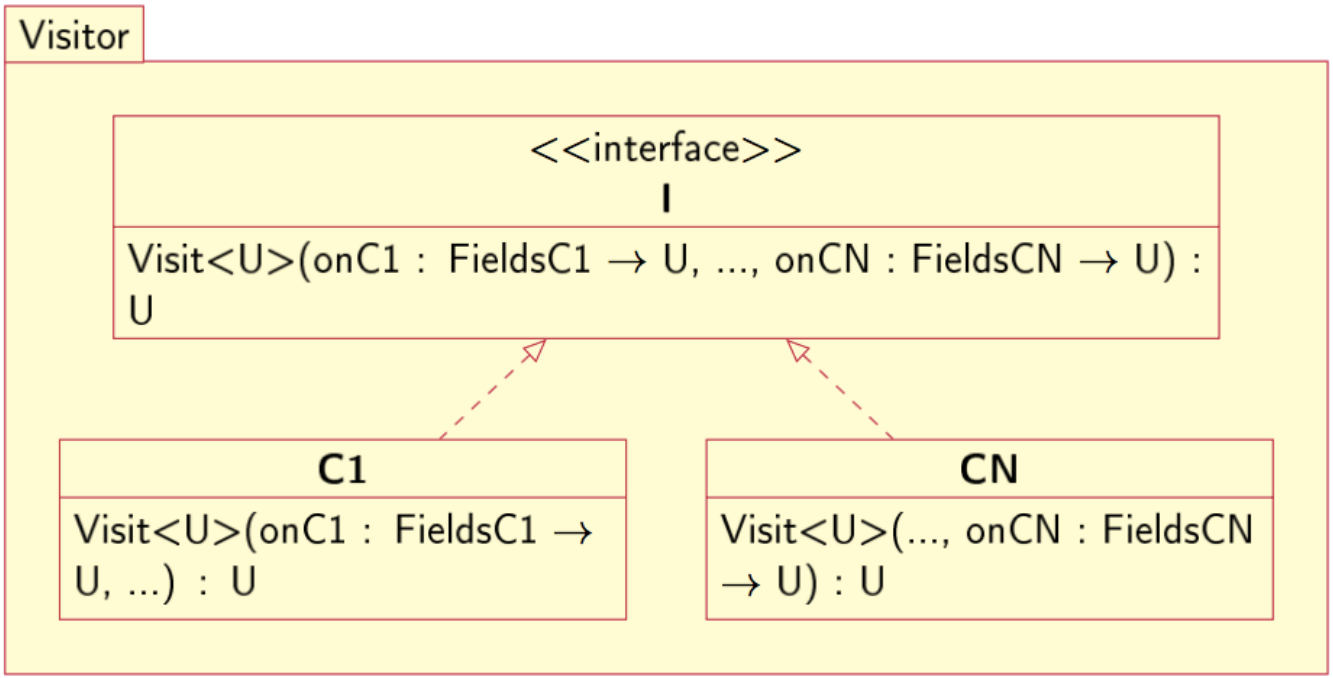
⏳ Lesson 8: Mock exam (June 20, 2017)

- Thanks for your attention!

YOUR QUESTIONS:

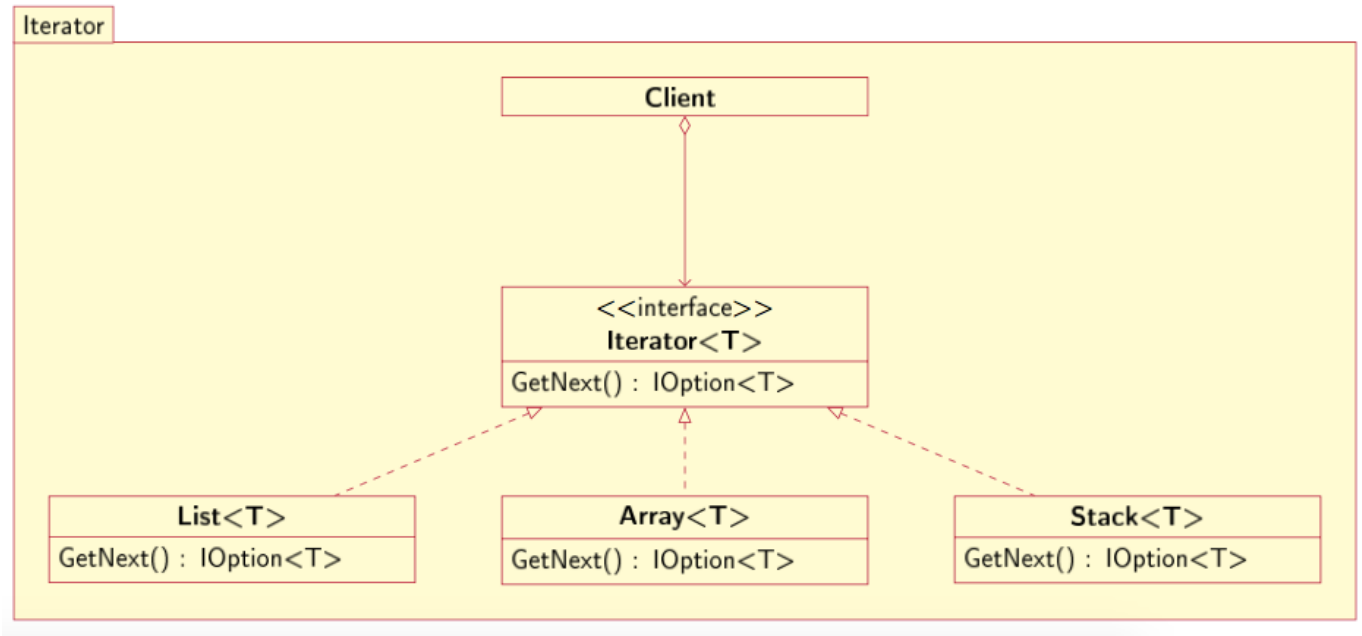
λ version of the Visitor

Intent: represent an operator to be performed on the elements of an object structure. Visitor allows a definition of a new operation without changing the classes of the elements on which it operates



λ version of the Iterator

Intent: a way to access elements of an aggregate object without exposing its underlying representation



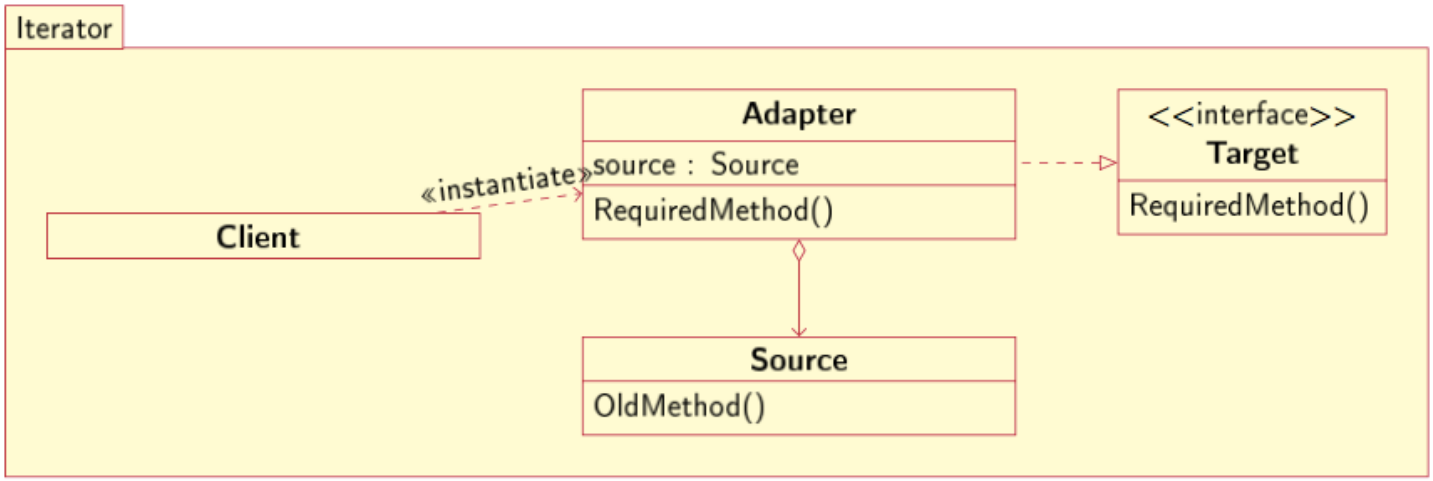
REVISING Λ NOTATION
ITERATOR: OFFICIAL SLIDES

REVISING Λ NOTATION
ADAPTER: OFFICIAL SLIDES

λ version of the Adapter

Intent: to convert the interface of a class into another interface that is expected by a client

(typo in the diagram below: title should read Adapter)



REVISING A NOTATION ABSTRACT FACTORY AND FACTORY METHOD PATTERNS: OFFICIAL SLIDES