COMP 3711

(OOA and OOD)

Iterations
And
GoF

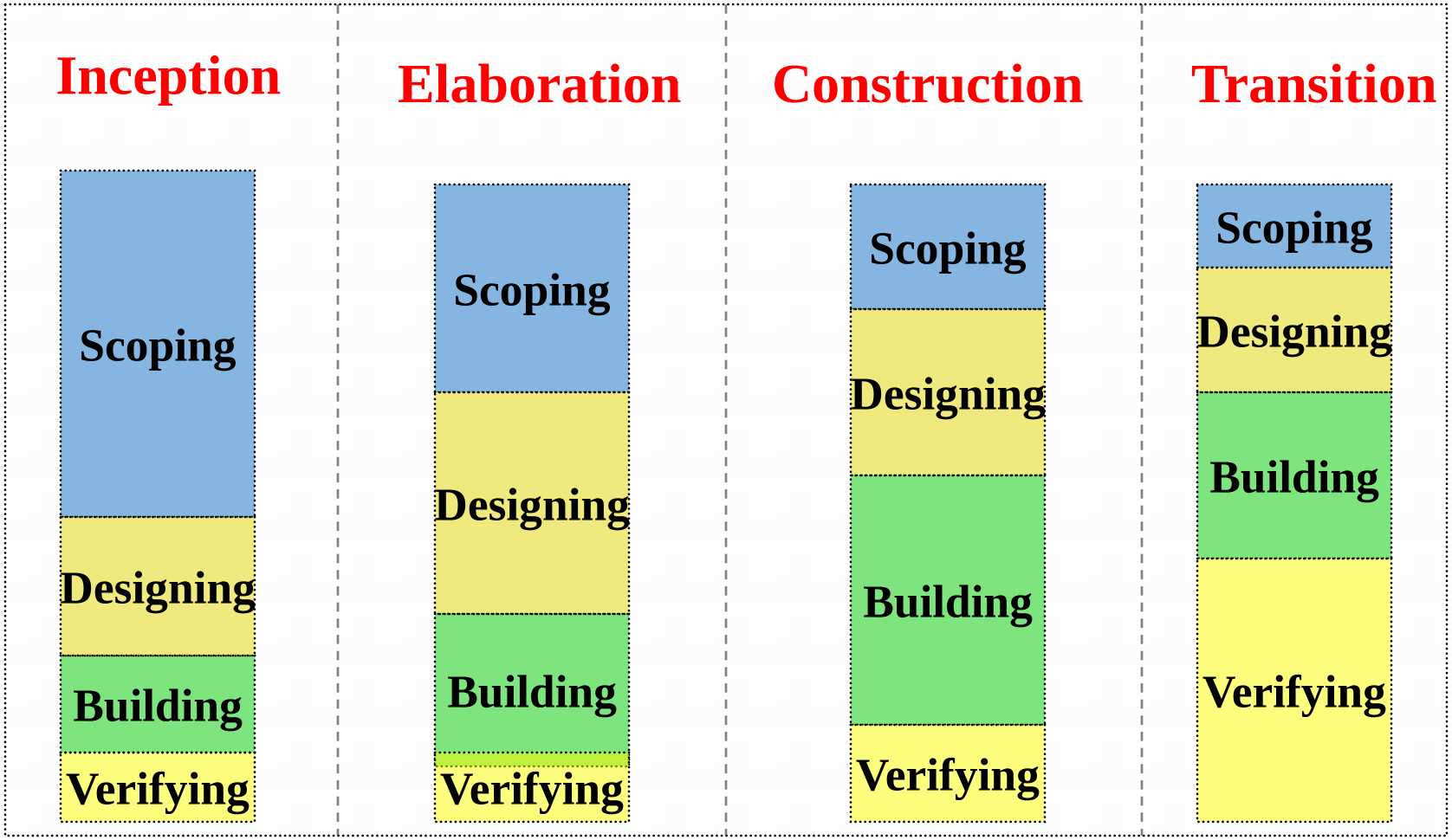
Larman Chapter 23, 26, 27

Unified Process - UP

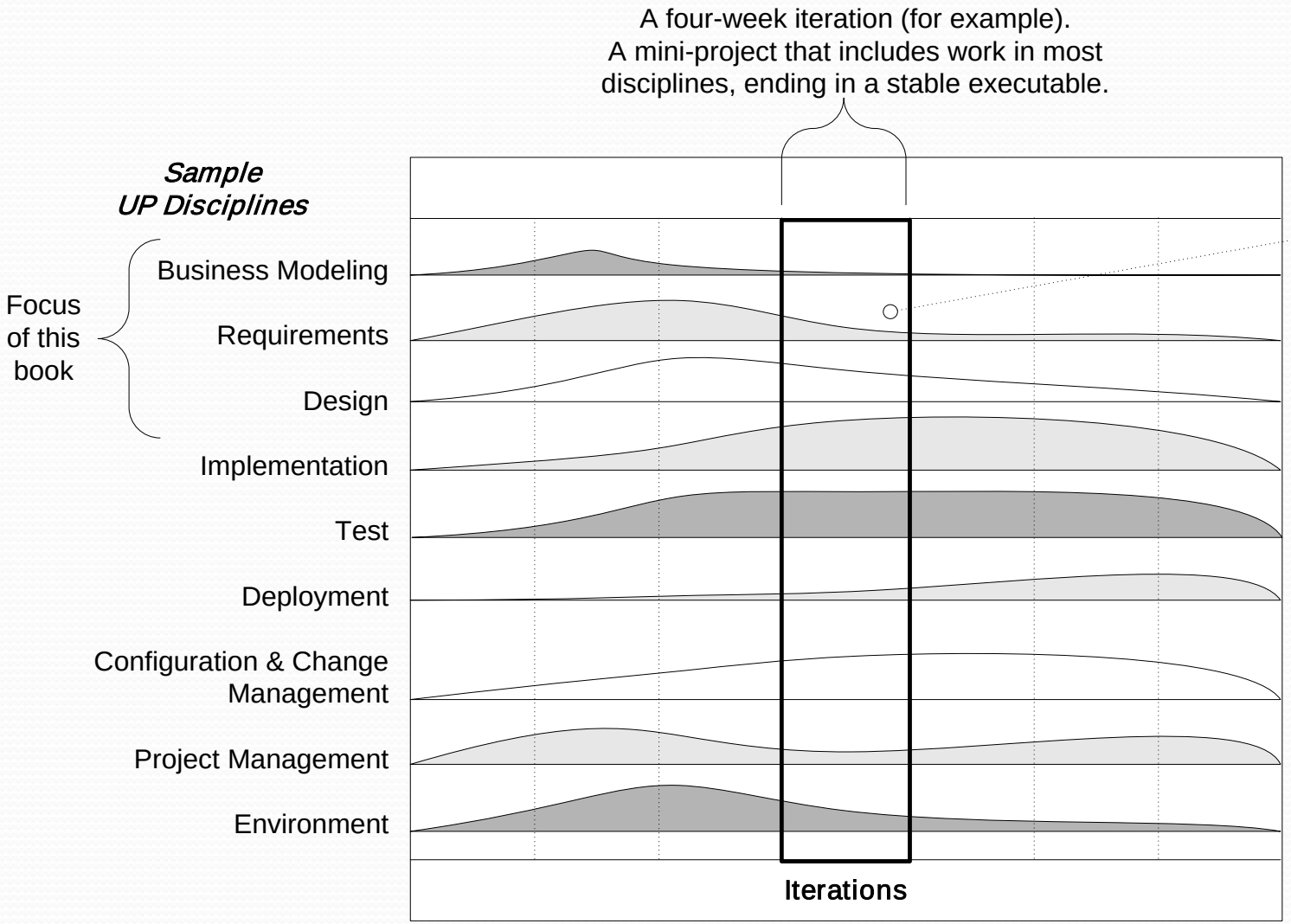
Requirement	Design	Develop	Implement
Inception	**Elaboration**	**Construction**	**Transition**
Approximate vision, business case, scope	Refine vision, core architecture, refine scope	Implementation of low risk core architecture, refine scope	Beta test and deployment

Iterative Development means not all requirements are implemented at once

Iterative UP



Iterative UP



Note that although an iteration includes work in most disciplines, the relative effort and emphasis change over time.

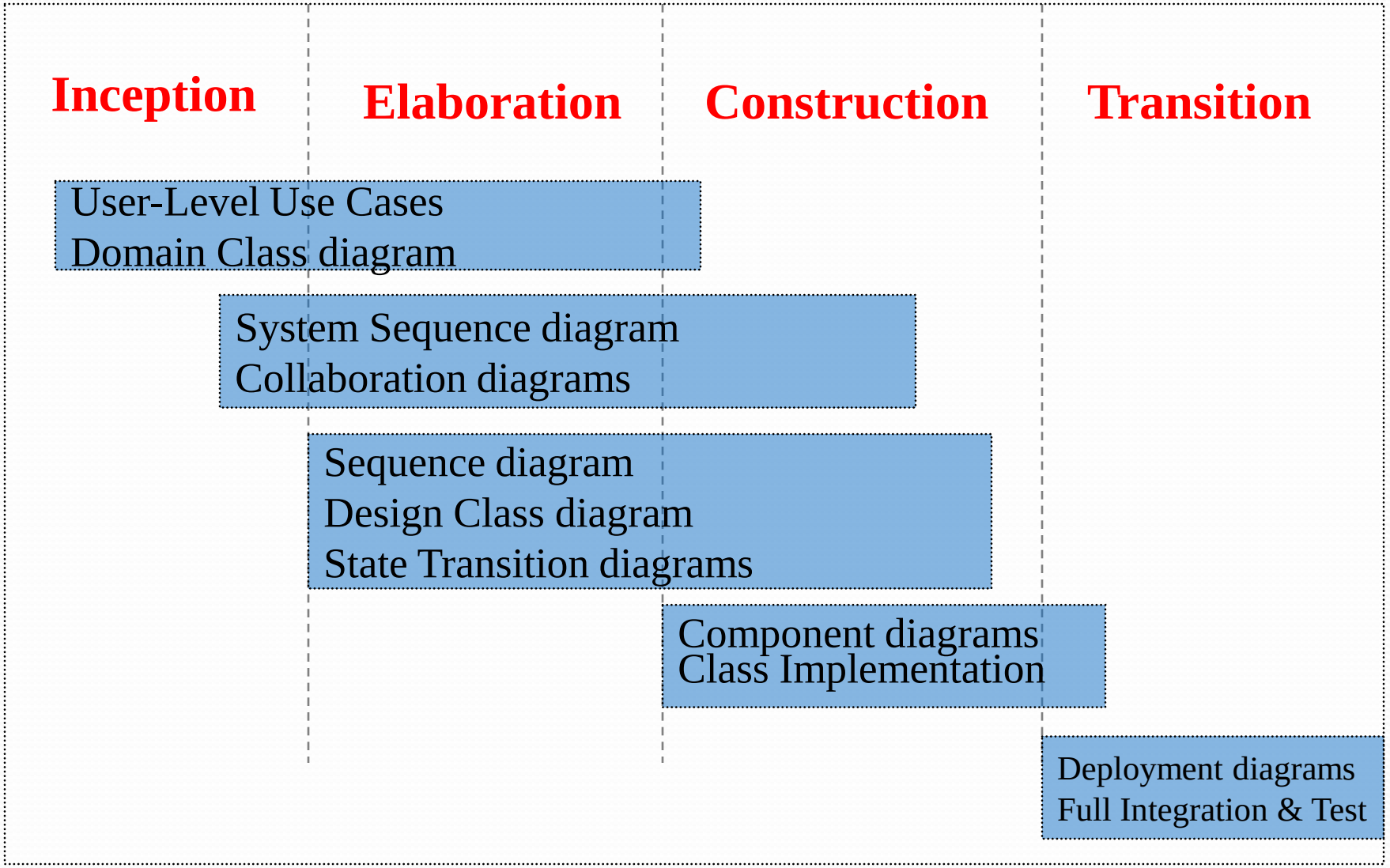
This example is suggestive, not literal.

Larman Fig 2.7

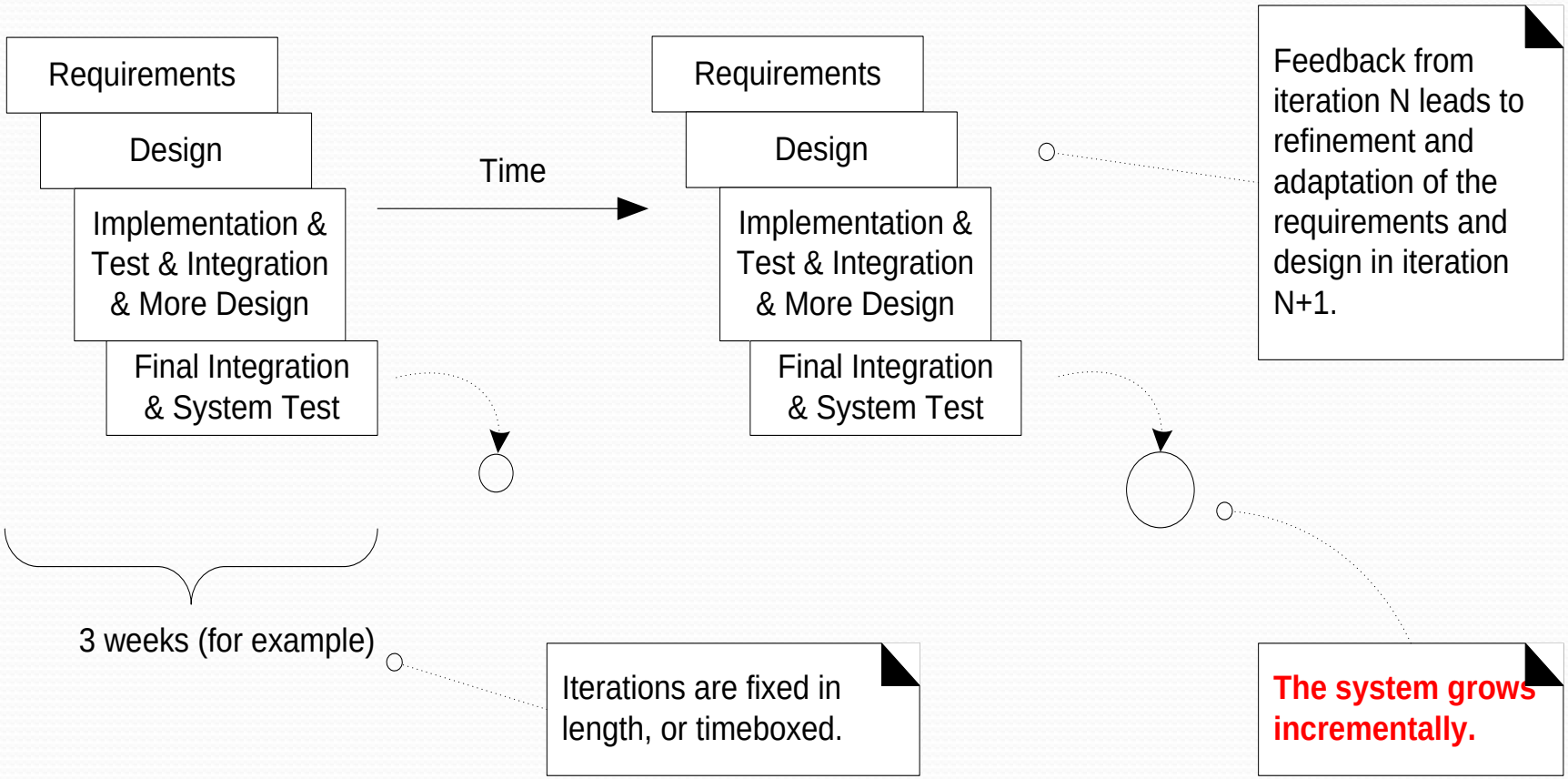
UP – Reinforces 6 Best Practices

- Develop iteratively
- Define and manage system requirements
- Use component architectures
- Create visual models
- Verify quality
- Control changes

UML Diagrams In UP Iterations



Iterative Development in UP



Larman fig 2.1

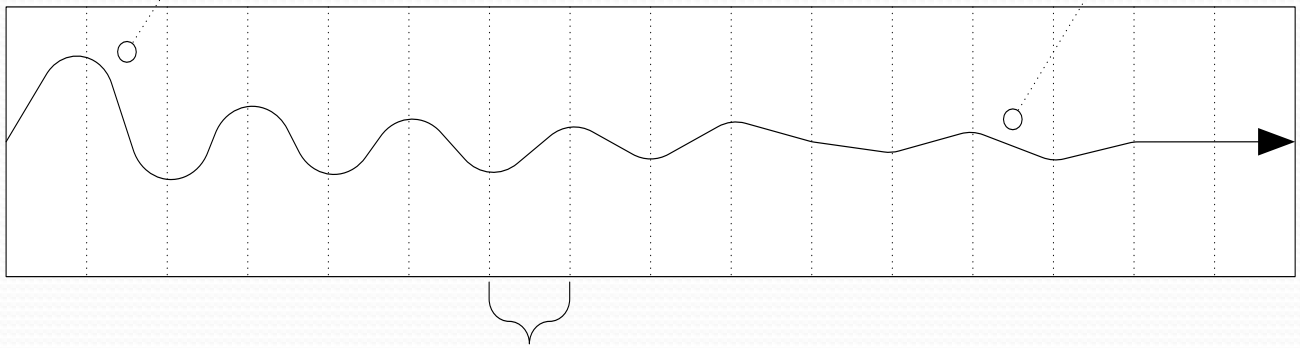
Iterative Development in UP

- Iteration 1
- Iteration 2
- Iteration 3
- Iteration .....

Iterative Development in UP

Early iterations are farther from the "true path" of the system. Via feedback and adaptation, the system converges towards the most appropriate requirements and design.

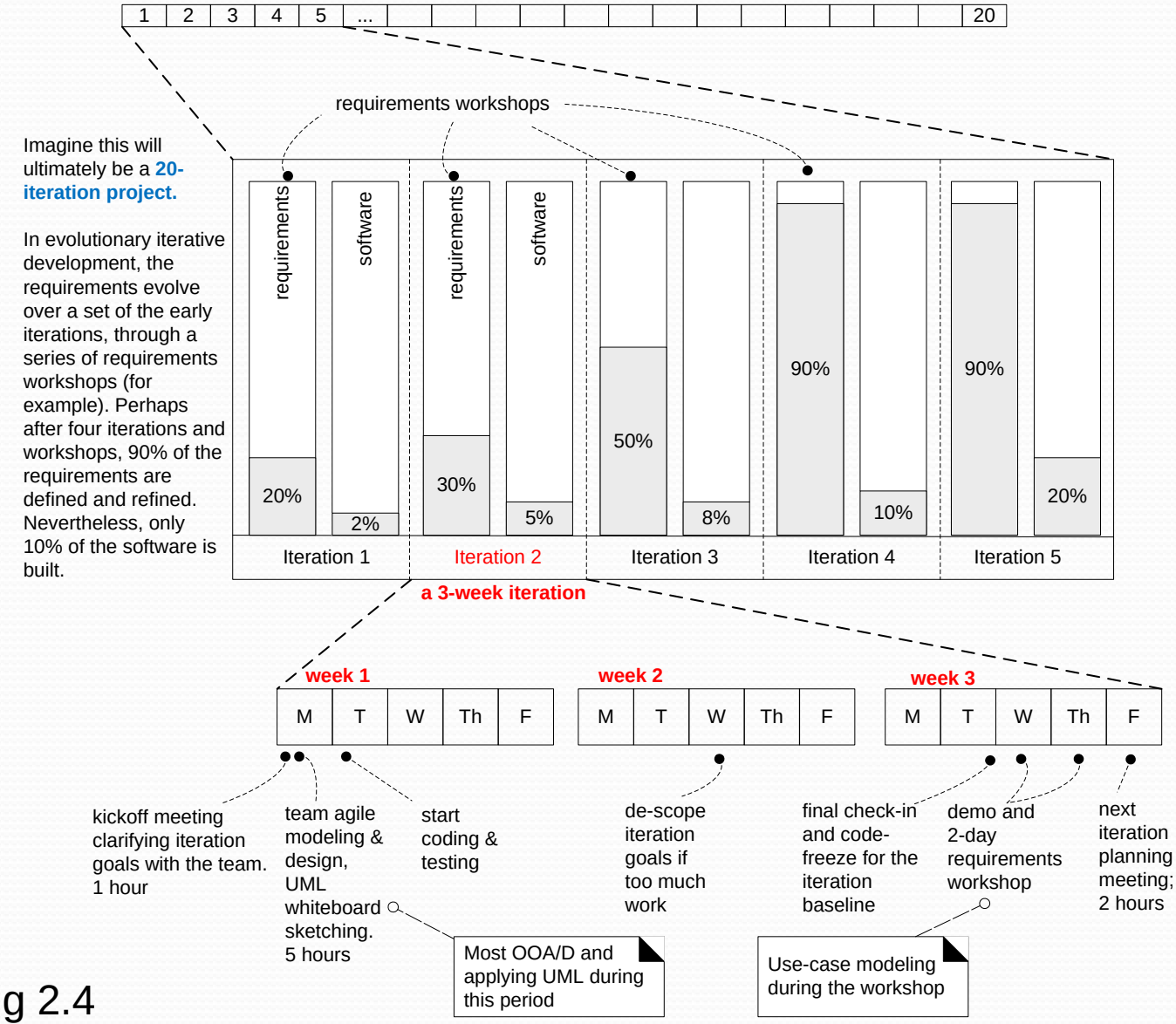
In **late iterations**, a significant change in requirements is rare, but can occur. Such late changes may give an organization a competitive business advantage.



one iteration of design, implement, integrate, and test

Larman fig 2.2

Iterative Development in UP



Larman fig 2.4

Iteration 1 - Accomplishment

- OOA (Domain Model, Use Case Model)
- OOD (Design Model)
- TDD
- Refactoring
- Database Model, Implementation Model
- In the 1st iteration, requirements chosen to be resolved are organized by risk and high business value (risk-driven and client-driven)

Subsequent Iterations

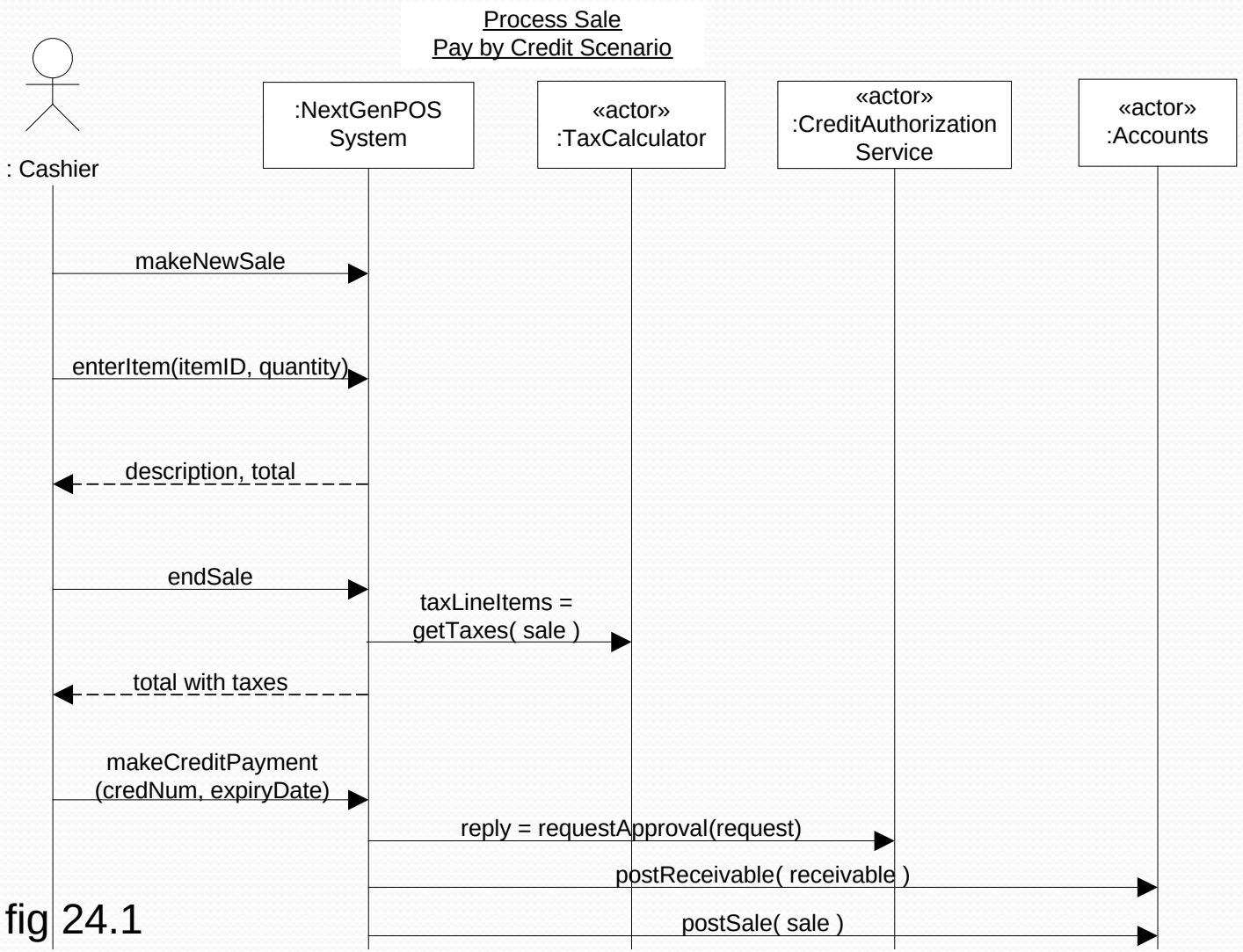
- Focus on OOA requirements already captured in Iteration 1 and apply RDD and GRASP / GoF design pattern to OOD

- Apply OOD incrementally to include more scenarios and more functionality (e.g. extension of the same iteration 1 use case)

NextGen POS Example

- Iteration 2
  - Example of a few additions to Iteration 1:
    - Support variations of 3rd party services
    - Complex pricing rules
    - Refresh GUI when sale total changes
  - What impact will the above have on Domain Model (if any)?
  - May be not necessary to refine the Domain Model when no significant values will be added

Iteration 2 – Update Iteration 1's SSD (Example: 3rd party services)



Larman fig 24.1

OO DESIGN - GoF PATTERNS

- For the purpose of our learning, in this next iteration for the development of NextGen POS, we will consider some GoF patterns in addition to GRASP

- Like GRASP, GoF patterns deal with recurring solutions to common problems in software design

- GoF – Gang-Of-Four are Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.

GoF Design Patterns Overview

"Design Patterns", mostly coded in C++ and Smalltalk , was introduced in 1994 by the Gang-of-Four, covering 23 patterns with 15 commonly used.

Creational	Structural	Behavioral
• Abstract • Factory • Builder • Prototype • Singleton	• Adapter • Bridge • Composite • Decorator • Facade • Flyweight • Proxy	• Chain of Responsibility • Command • Interpreter • Iterator • Mediator • Momento • Observer • State • Strategy • Template Method • Visitor

See http://en.wikipedia.org/wiki/Design_Patterns

The GoF Patterns

- Creational
  - Deal with class instantiation
- Structural
  - Deal with Class and Object composition
- Strategy
  - Deal with communication between objects

A Few Selected GoF Patterns For NextGen POS Example

- Adapter
- Factory
- Singleton
- Strategy
- Composite

GoF *adaptor* pattern

Name: Adaptor

Problem

- How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?

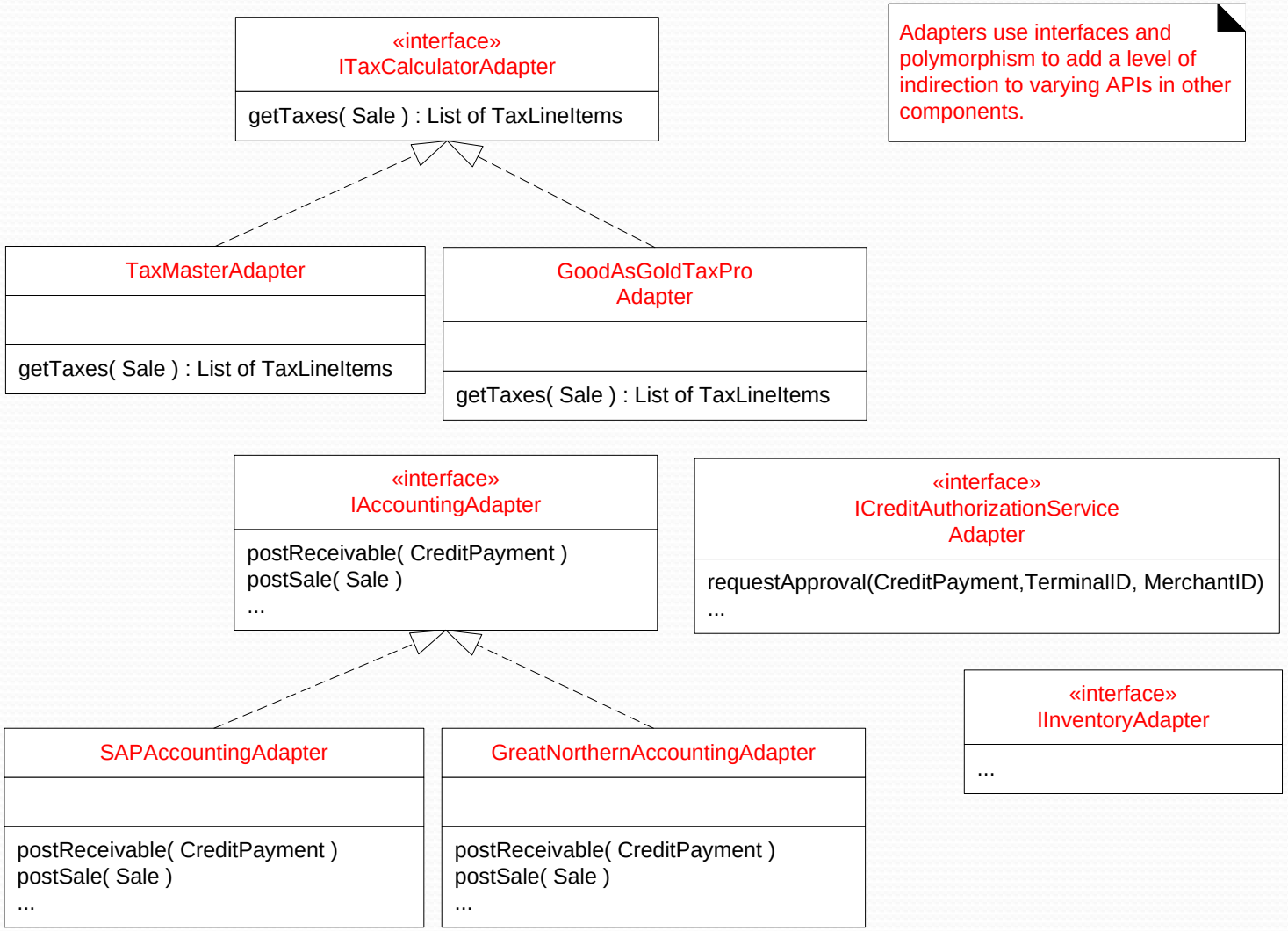
Solution

- Convert the original interface of a component into another interface, through an intermediate adapter object

GoF *adaptor* pattern Advantages

- As in GRASP, GoF *adaptor* provides *Protected Variations* from changing external interfaces or third party packages through the use of an *Indirection* object that applies interfaces and *Polymorphism*

Iteration 2 – add GoF *adaptor* pattern



Larman fig 26.1

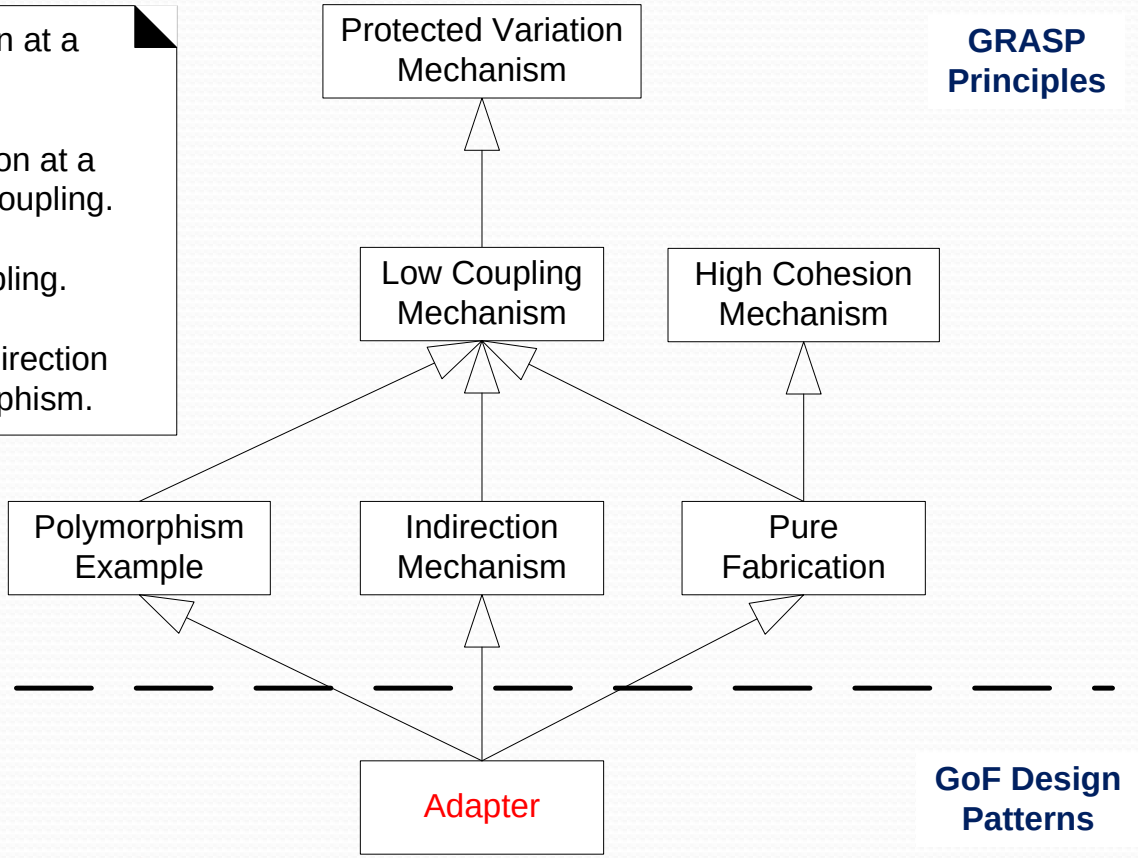
GRASP vs GoF *adaptor* pattern

Low coupling is a way to achieve protection at a variation point.

Polymorphism is a way to achieve protection at a variation point, and a way to achieve low coupling.

An indirection is a way to achieve low coupling.

The Adapter design pattern is a kind of Indirection and a Pure Fabrication, that uses Polymorphism.



Who creates the adaptor?

Larman fig 26.3

GoF *factory* pattern

Name: Factory

Problem

- Who should be responsible for creating objects when there are special considerations, such as complex creation logic, desire to separate creation responsibilities for better cohesion?
- How to determine which class of adapter to create?

Solution

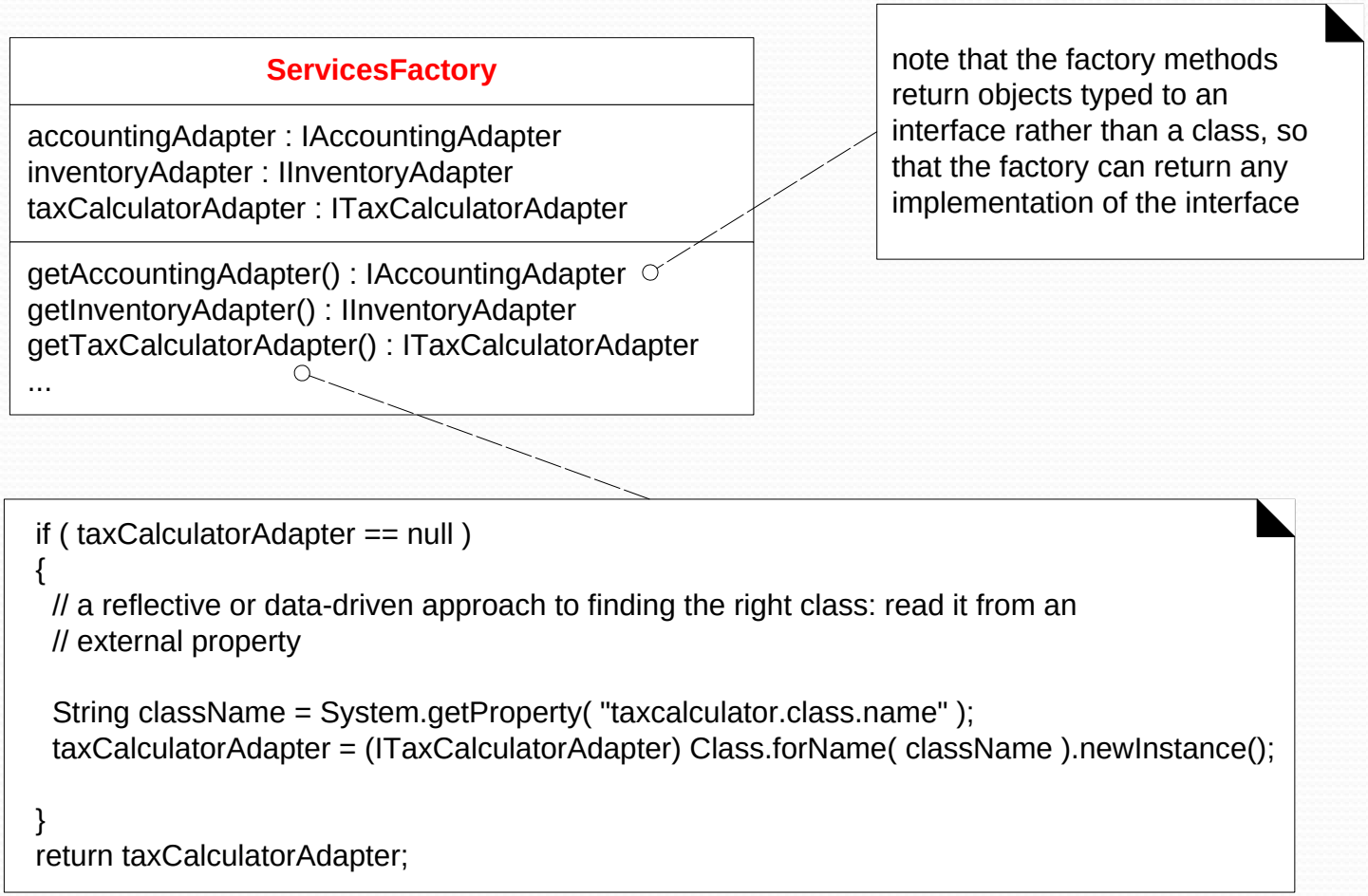
- Create a Pure Fabrication object called a *Factory* that handles the creation

GoF *factory* pattern Advantages

- Design to separate responsibility of complex creation into cohesive helper objects
- Hide potentially complex creation logic
- Allow performance enhancing memory management strategies ,such as caching or recycling

Iteration 2 – add *Factory* pattern

In order to maintain high cohesion - apply a pure fabrication *Factory* pattern to create objects (such as adaptors)



Larman fig 26.5

GoF *singleton* pattern

Name: Singleton

Problem

- Just one instance of the class is allowed – “singleton”
- Objects need one point of access and global data

Solution

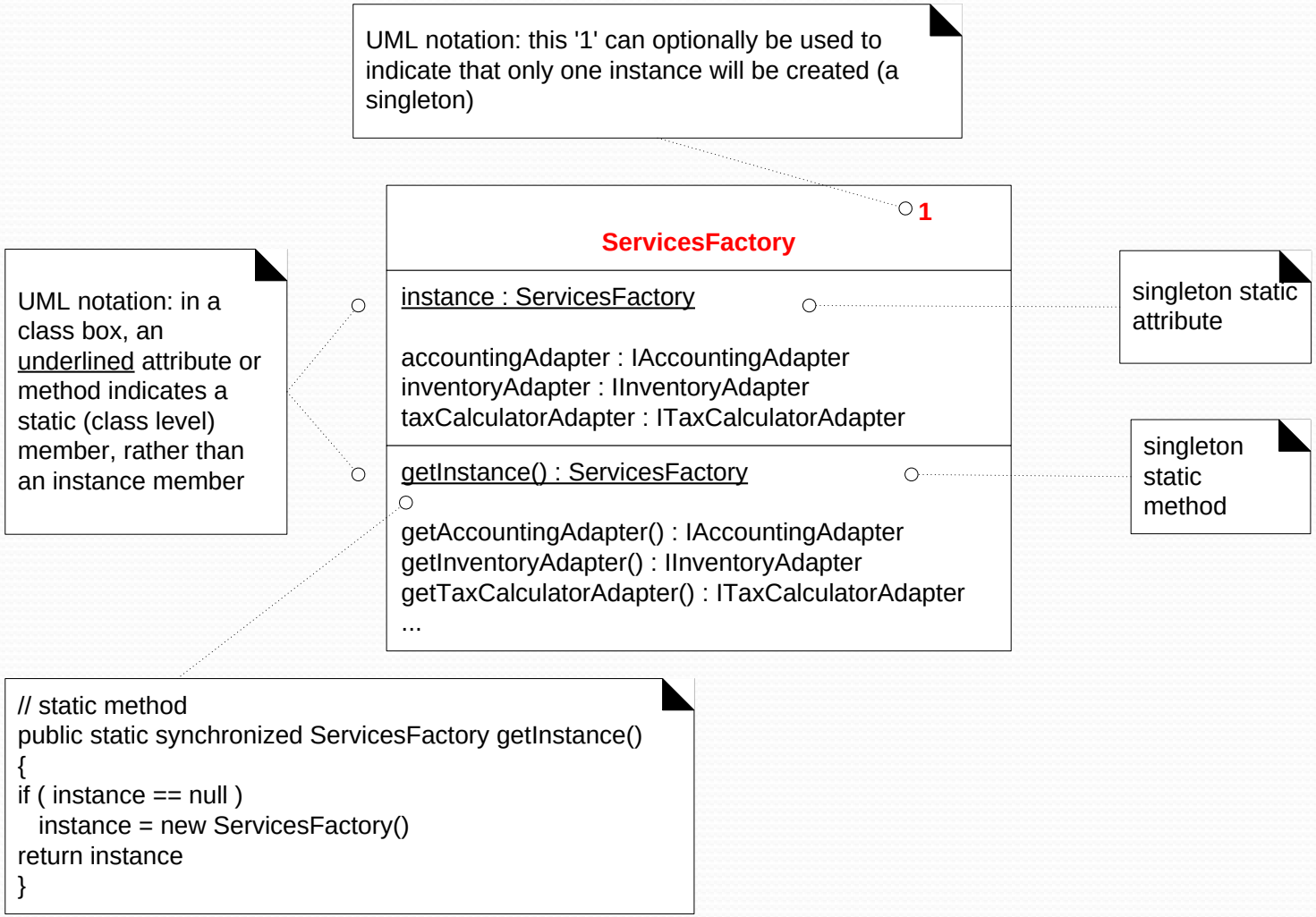
- Define a static method of the class that returns the singleton

GoF *singleton* pattern Advantages

- Controlled access to a single instance
- *Intance-side methods permit subclassing and refinement of Singleton* class into subclasses
- Global access without a global variable

Iteration 2 – add GoF *Singleton* pattern

Apply a singleton pattern to get visibility to create the Factory class (only one instance is needed)



Larman fig 26.6

GoF *strategy* pattern

- Name: Strategy

- Problem
  - How to design for varying but related algorithms (strategies) or policies?
  - How to design for the ability to change these algorithms or policies?

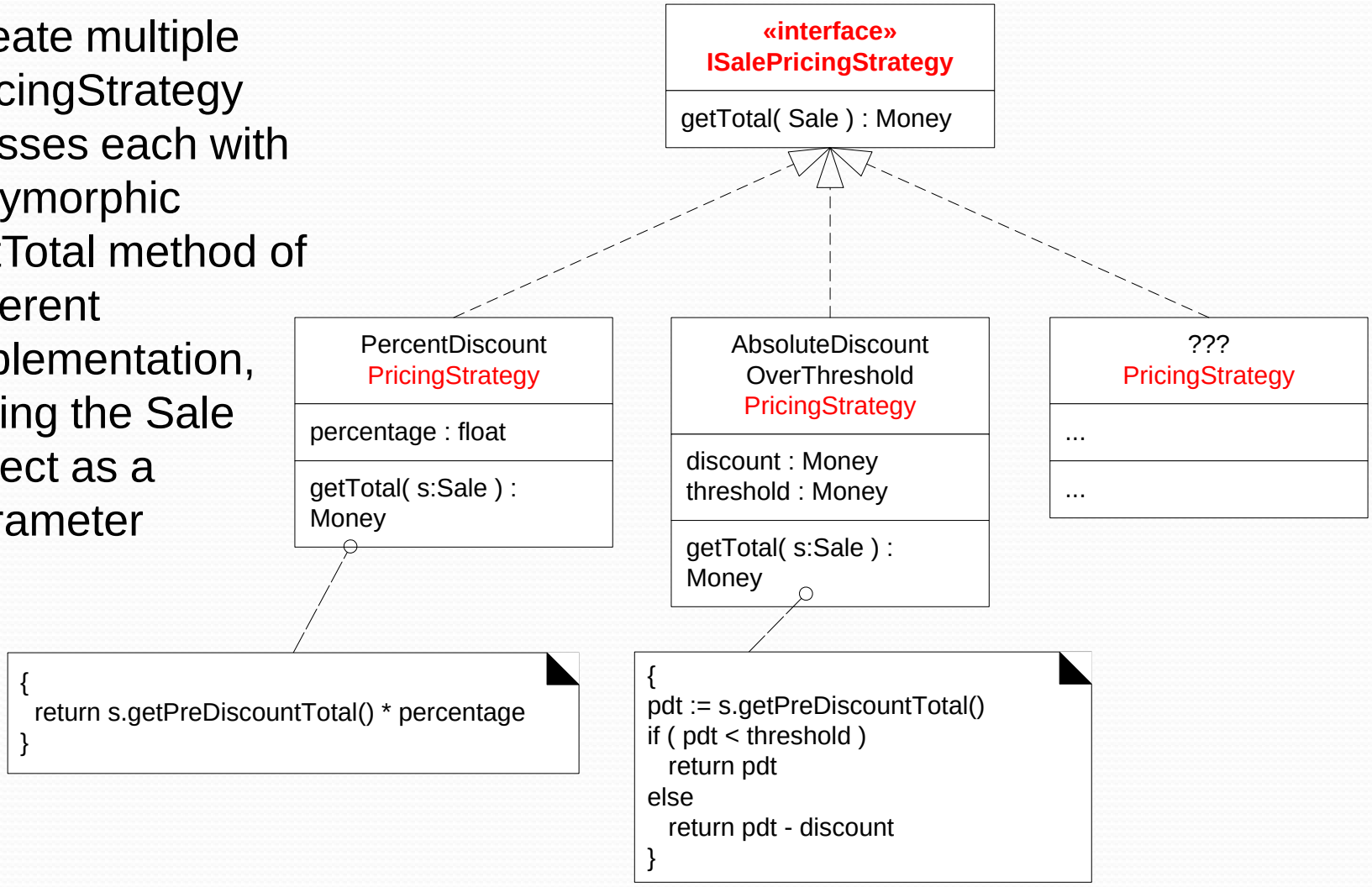
- Solution
  - Define each policy or algorithm in a separate class and have a common interface
  - Attach a strategy object to a context object

GoF *strategy* pattern Advantages

- Flexible alternative to sub-classing
  - Separates algorithms from context
- Eliminates conditional statements
- Polymorphism
  - Can implement same behavior in different ways

Iteration 2 – add GoF *strategy* pattern

Create multiple PricingStrategy classes each with polymorphic getTotal method of different implementation, taking the Sale object as a parameter



Larman fig 26.9

GoF composite pattern

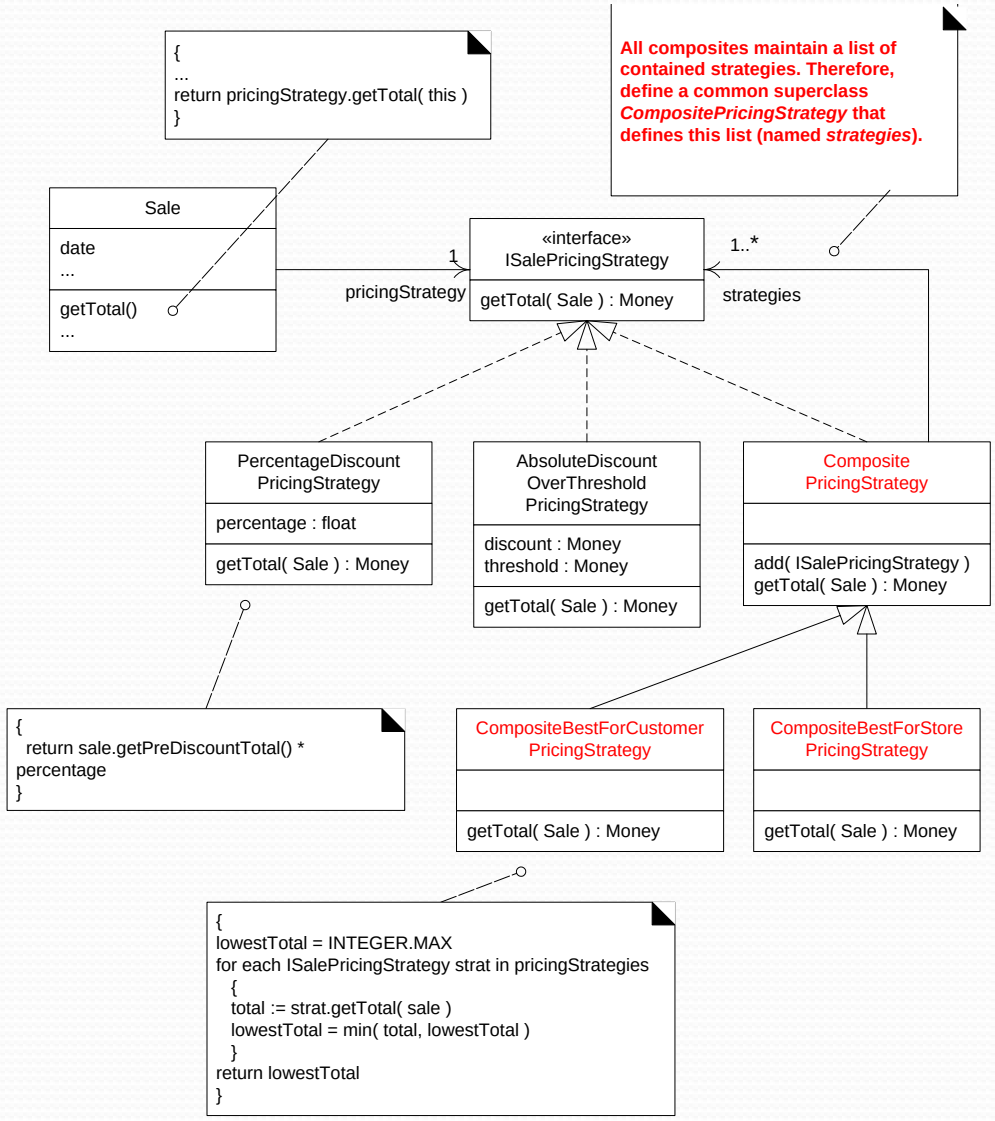
- Name: Composite
- Problem
  - How to treat a group or composition structure of objects the same way (polymorphically) as a non-composite (atomic) object?
- Solution
  - Define classes for composite and atomic objects so that they implement the same interface

GoF *composite* pattern Advantages

- Help to resolve conflicting strategies
- Flexible to combine the use of *atomic strategy* objects and *composite* strategy objects without conflict
- Context object does not need to know detail implementation

Iteration 2 – add GoF *composite* pattern

- There may exist multiple conflicting co-existing strategies, i.e. one sale may have varying Pricing strategies based on:
  - time period
  - customer type
  - specific line item



Larman fig 26.14