23rd Summer School on PARALLEL COMPUTING

Design patterns: an introduction

Paolo Ciancarini – paolo.ciancarini@unibo.it
Department of Informatics – University of Bologna

CINECA SCAI
SuperComputing Applications and Innovation

cini consorzio interuniversitario nazionale per l'informatica

Motivation and Concept

- Software systems should exploit reusable design knowledge promoting
  - Abstraction
  - Flexibility
  - Modularity
  - Elegance
- Problem: capturing, communicating, and applying design knowledge

Patterns for HPC

- Which technologies improve the productivity of HPC software development?
- parallel programming languages and libraries, Object Oriented scientific programming, and parallel run-time systems and tools.
- The success of these activities requires a good understanding of common patterns used in the development of HPC software:
- patterns used for the coding of parallel algorithms, their mapping to various architectures, and performance tuning activities.

What Is a Design Pattern?

- A design pattern
  - Is a common solution to a recurring problem in design
  - Abstracts a recurring design structure
  - Comprises class and/or object
    - Dependencies
    - Structures
    - Interactions
    - Conventions
  - Names & specifies the design structure explicitly
  - Distils design experience

What Is a Design Pattern?

- A design pattern has 4 basic parts:
  - Name
  - Problem
  - Solution
  - Consequences and trade-offs of application
- Language- and implementation-independent
- A “micro-architecture”
- Adjunct to existing methodologies (Unified, OMT, etc.)
- No mechanical application
  - The solution needs to be translated into concrete terms in the application context by the developer

Design Pattern Catalogues

- GoF ("the gang of four") catalogue
  - "Design Patterns: Elements of Reusable Object-Oriented Software," Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995
- POSA catalogue
  - Pattern-Oriented Software Architecture, Buschmann, et al.; Wiley, 1996
- …

Design Space for GoF Patterns

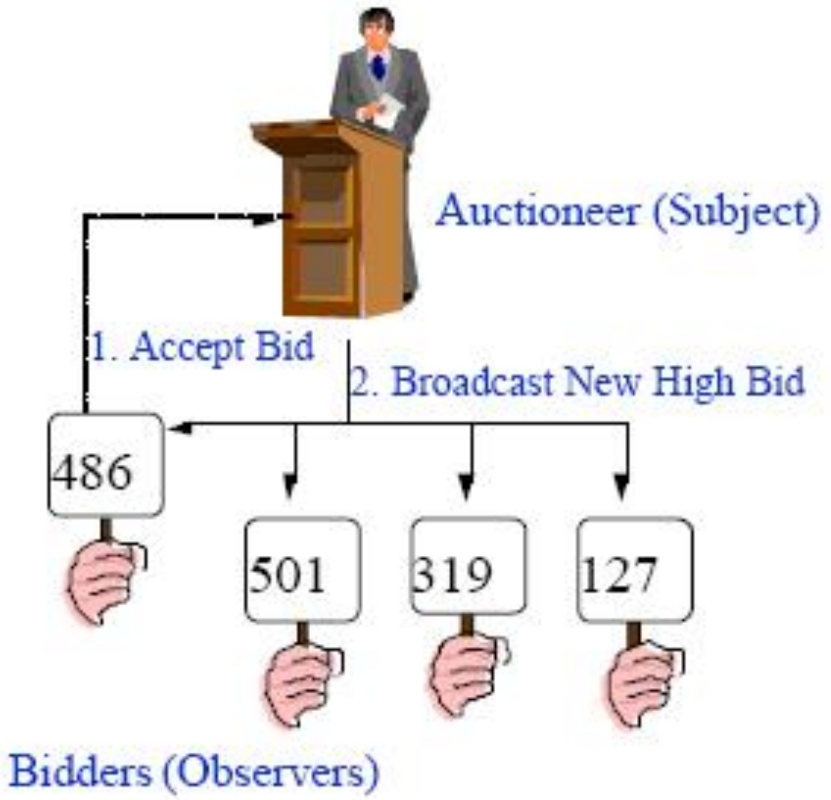
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Scope: domain over which a pattern applies
Purpose: reflects what a pattern does

Observer (Behavioral)

- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Applicability
  - When an abstraction has two aspects, one dependent on the other
  - When a change to one object requires changing others, and you don't know how many objects need to be changed
  - When an object should notify other objects without making assumptions about who these objects are

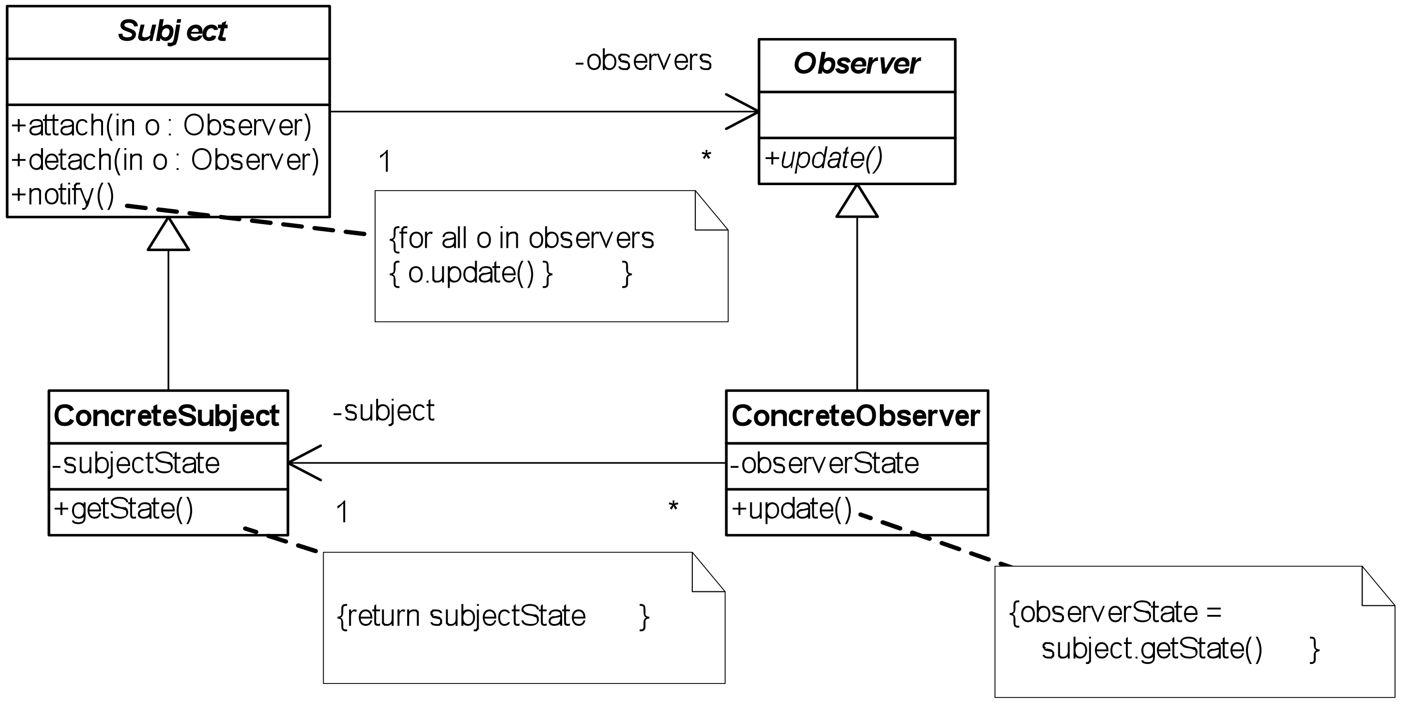
Observer



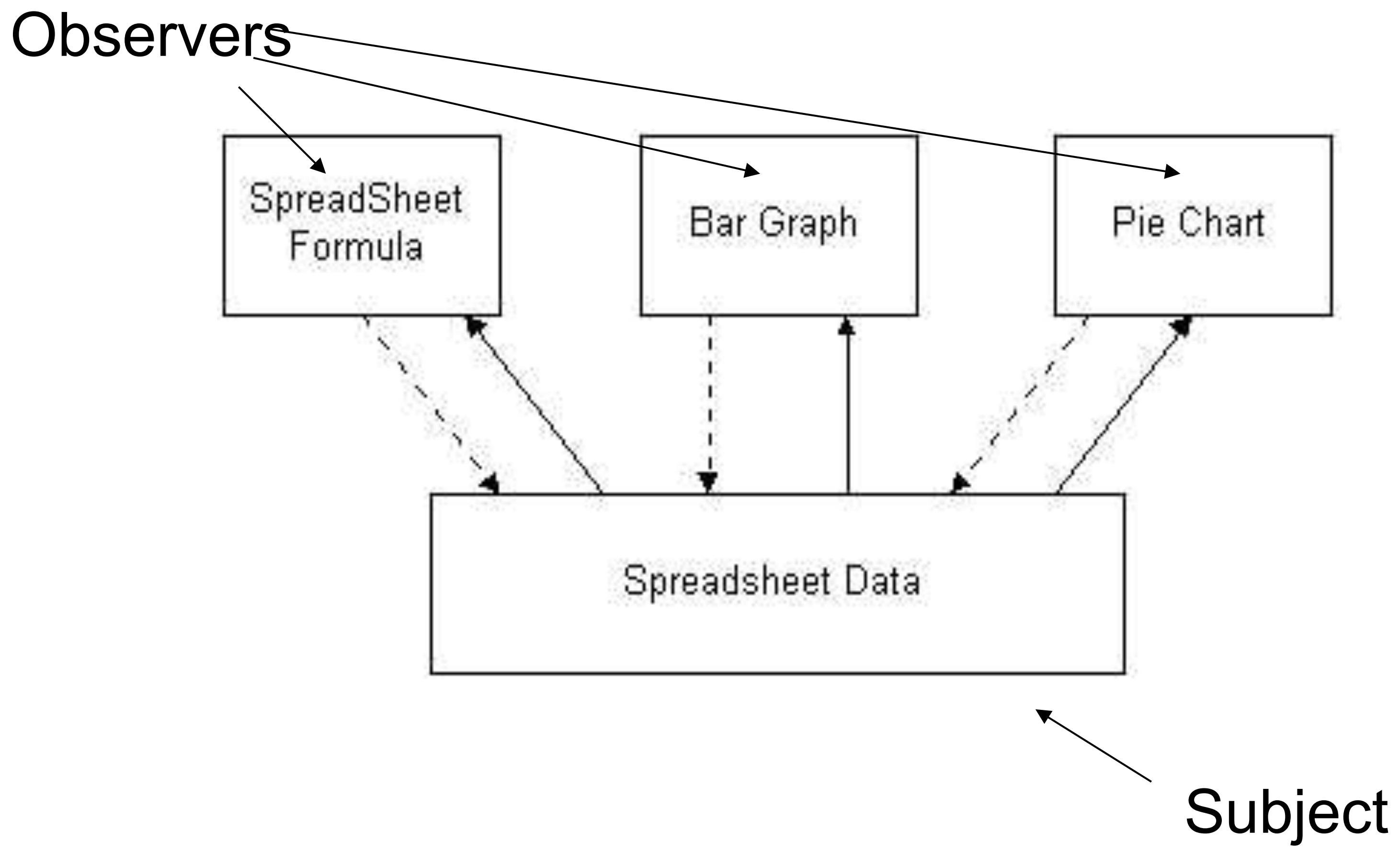
When a bidder at an auction accepts a bid, he or she raises a numbered paddle which identifies the bidder. The bid price then changes and all *Observers* must be notified of the change. The auctioneer then broadcasts the new bid to the bidders.

Behavioral

Observer: class diagram



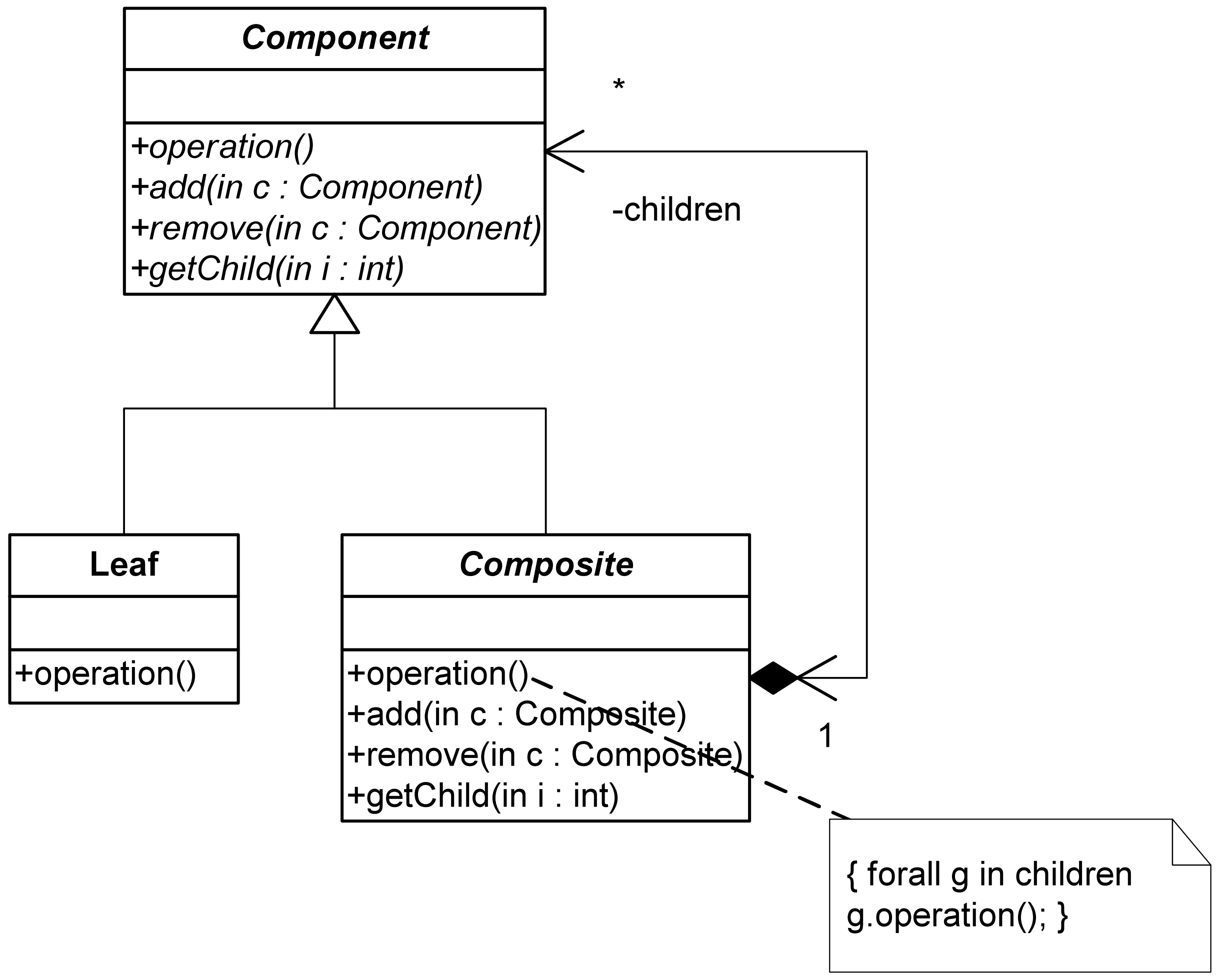
Observer: example



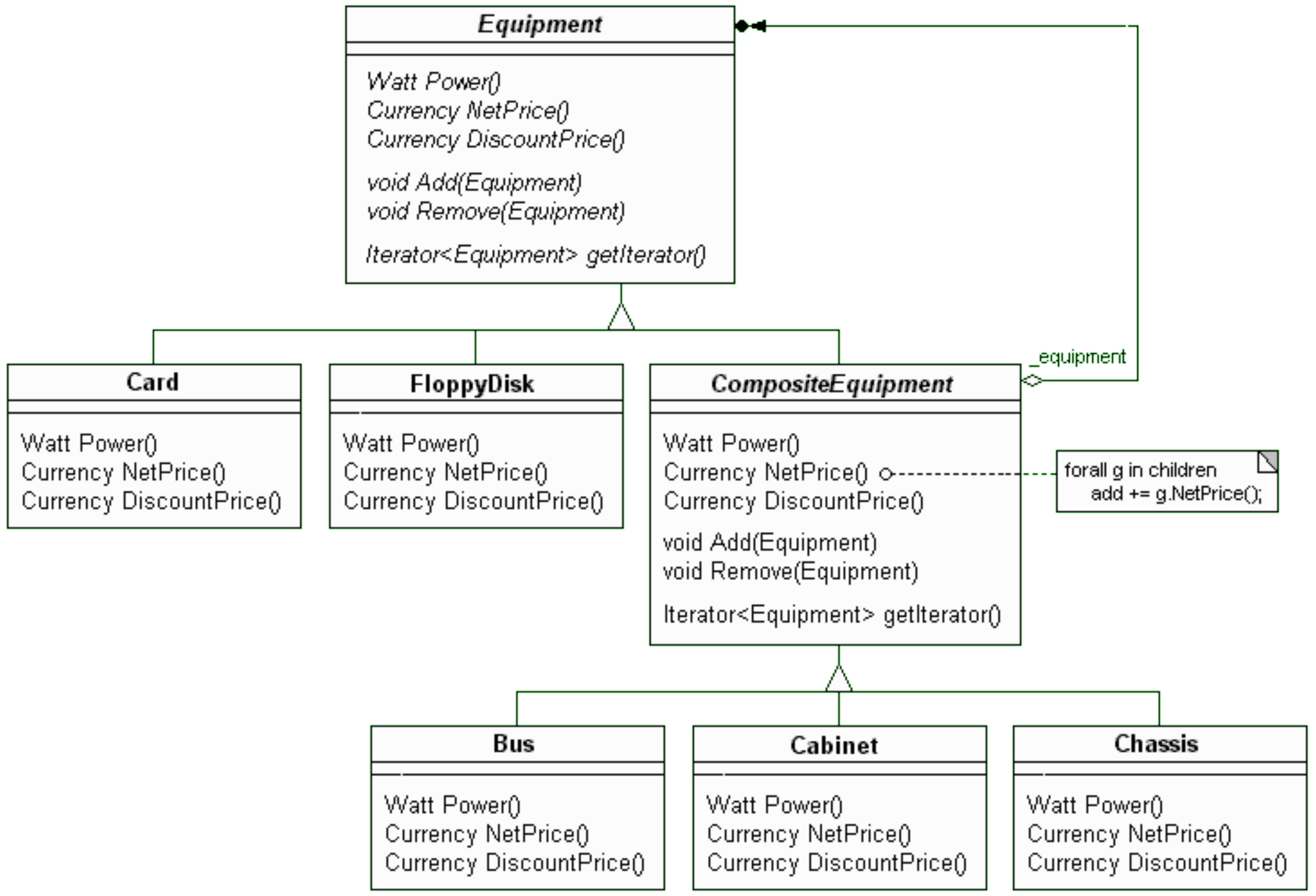
Composite (Structural)

- Intent
  - Treat individual objects and multiple, recursively-composed objects uniformly
- Applicability
  - Objects must be composed recursively,
  - And there should be no distinction between individual and composed elements,
  - And objects in the structure can be treated uniformly
  - Part-Whole hierarchy of objects.
  - Constant handling of objects as groups or individuals

Composite: class diagram



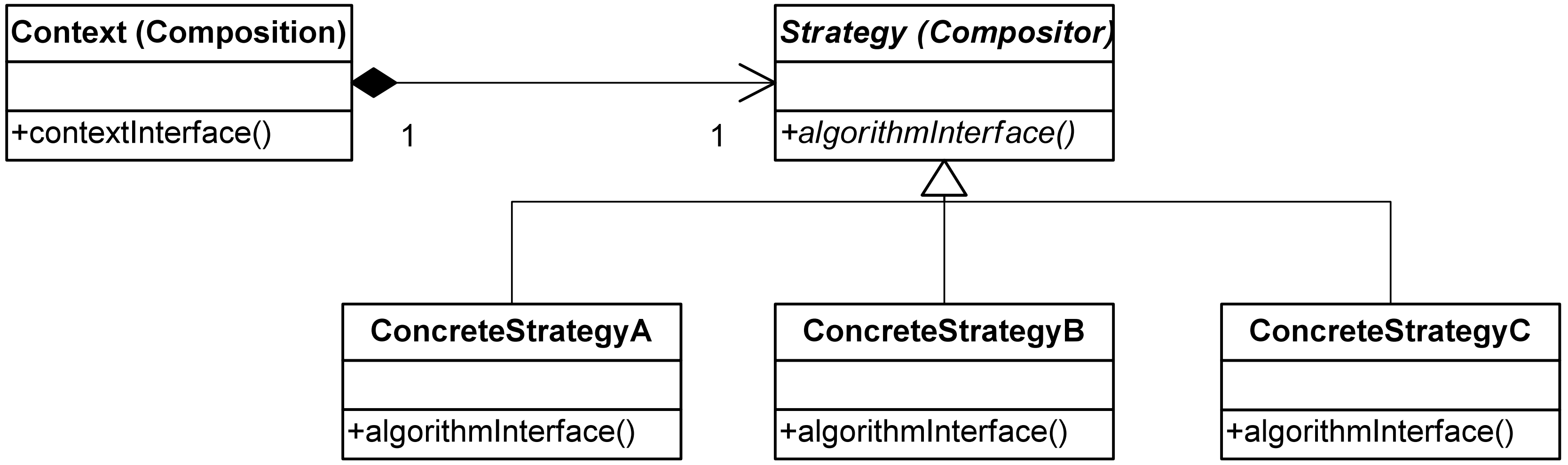
Composite: example



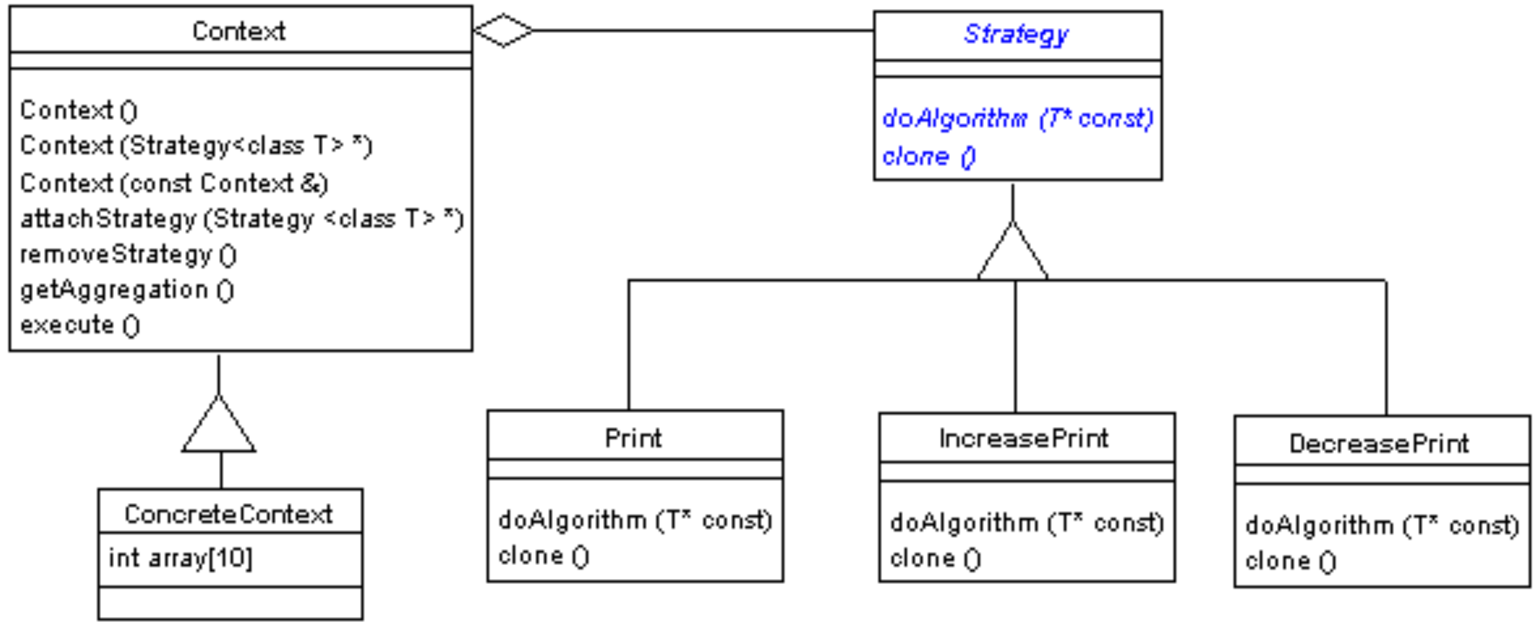
Strategy (Behavioral)

- Intent
  - Define a family of algorithms, encapsulate each one, and make them interchangeable to let clients and algorithms vary independently
- Applicability
  - When an object should be configurable with one of several algorithms,
    and all algorithms can be encapsulated,
    and one interface covers all encapsulations

Strategy: class diagram



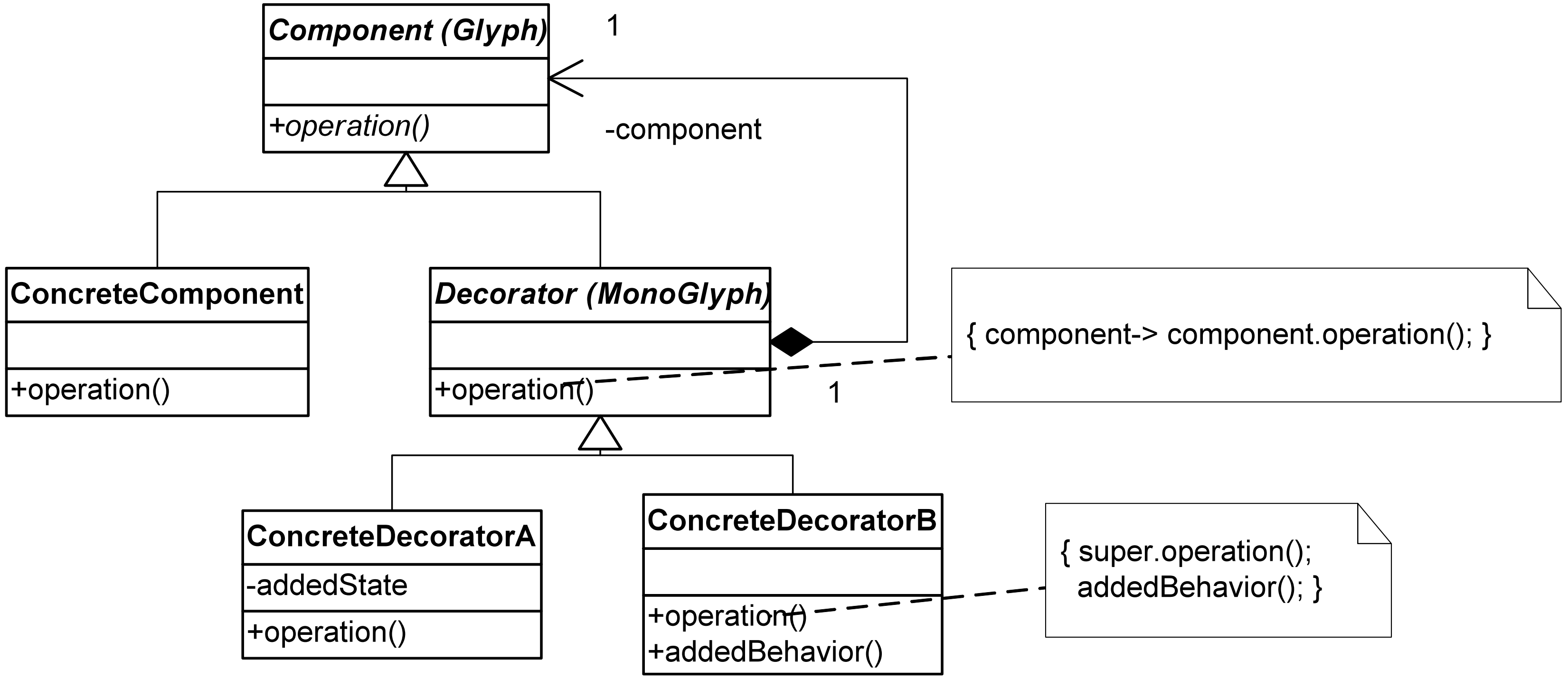
Strategy: example



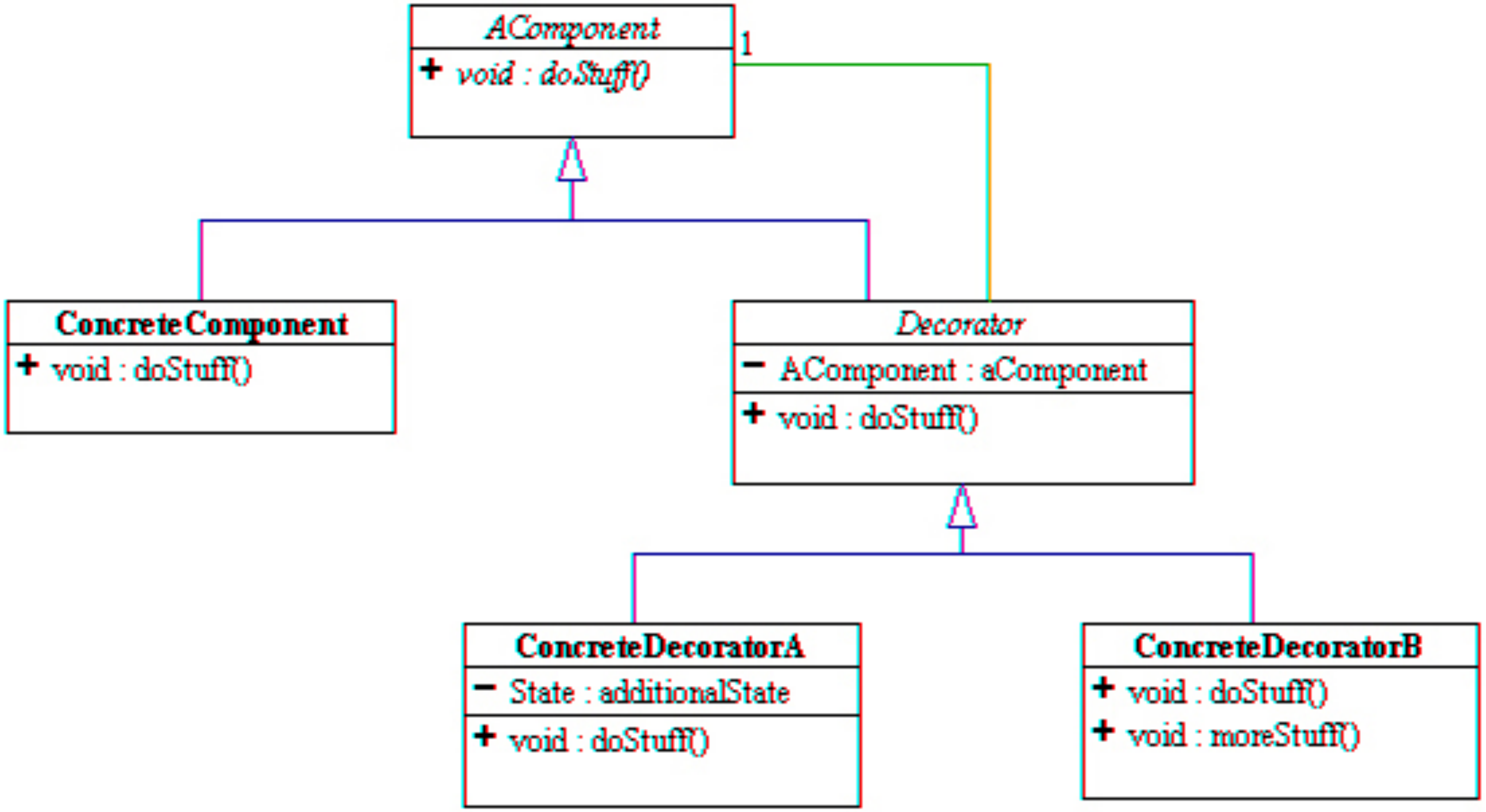
Decorator (Structural)

- Intent
  - Augment objects with new responsibilities
- Applicability
  - When extension by subclassing is impractical
  - For responsibilities that can be withdrawn

Decorator: class diagram



Decorator - Diagram



Decorator overview

- A Decorator, also known as a Wrapper, is an object that has an interface identical to an object that it contains. Any calls that the decorator gets, it relays to the object that it contains, and adds its own functionality along the way, either before or after the call
- Therefore, the Decorator Pattern is used for adding additional functionality to a particular object as opposed to a class of objects
- It is easy to add functionality to an entire class of objects by subclassing an object, but it is impossible to extend a single object this way. With the Decorator Pattern, you can add functionality to a single object and leave others like it unmodified

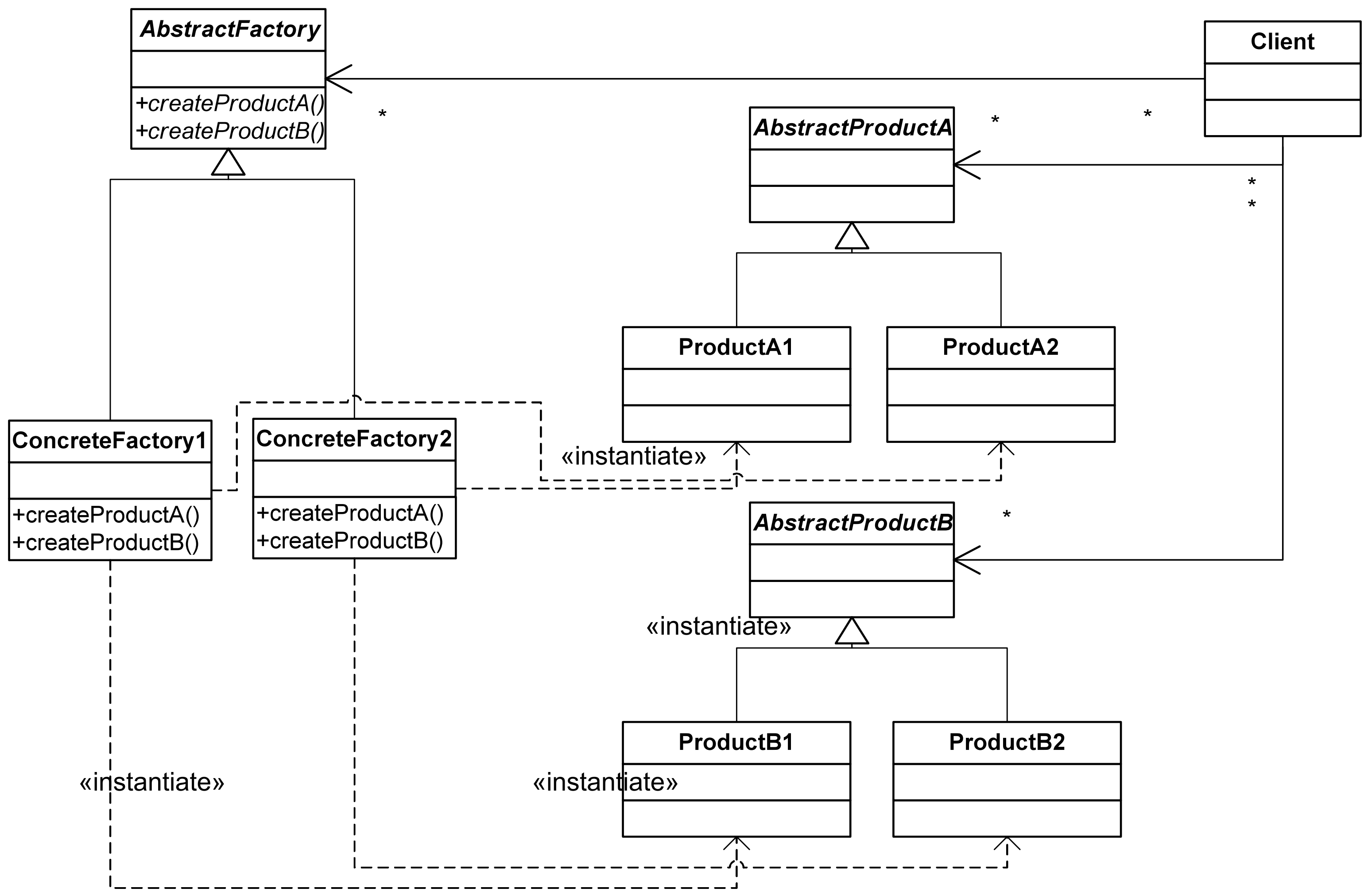
Decorator comments

- The Decorator pattern offers a lot of flexibility, since you can change what the decorator does at runtime, as opposed to having the change be static and determined at compile time by subclassing
- Since a Decorator complies with the interface that the object that it contains, the Decorator is indistinguishable from the object that it contains. That is, a Decorator is a concrete instance of the abstract class, and thus is indistinguishable from any other concrete instance, including other decorators
- This can be used to great advantage, as you can recursively nest decorators without any other objects being able to tell the difference, allowing a near infinite amount of customization

Abstract Factory (Creational)

- Intent
  - Create families of related objects without specifying class names
- Applicability
  - When clients cannot anticipate groups of classes to instantiate

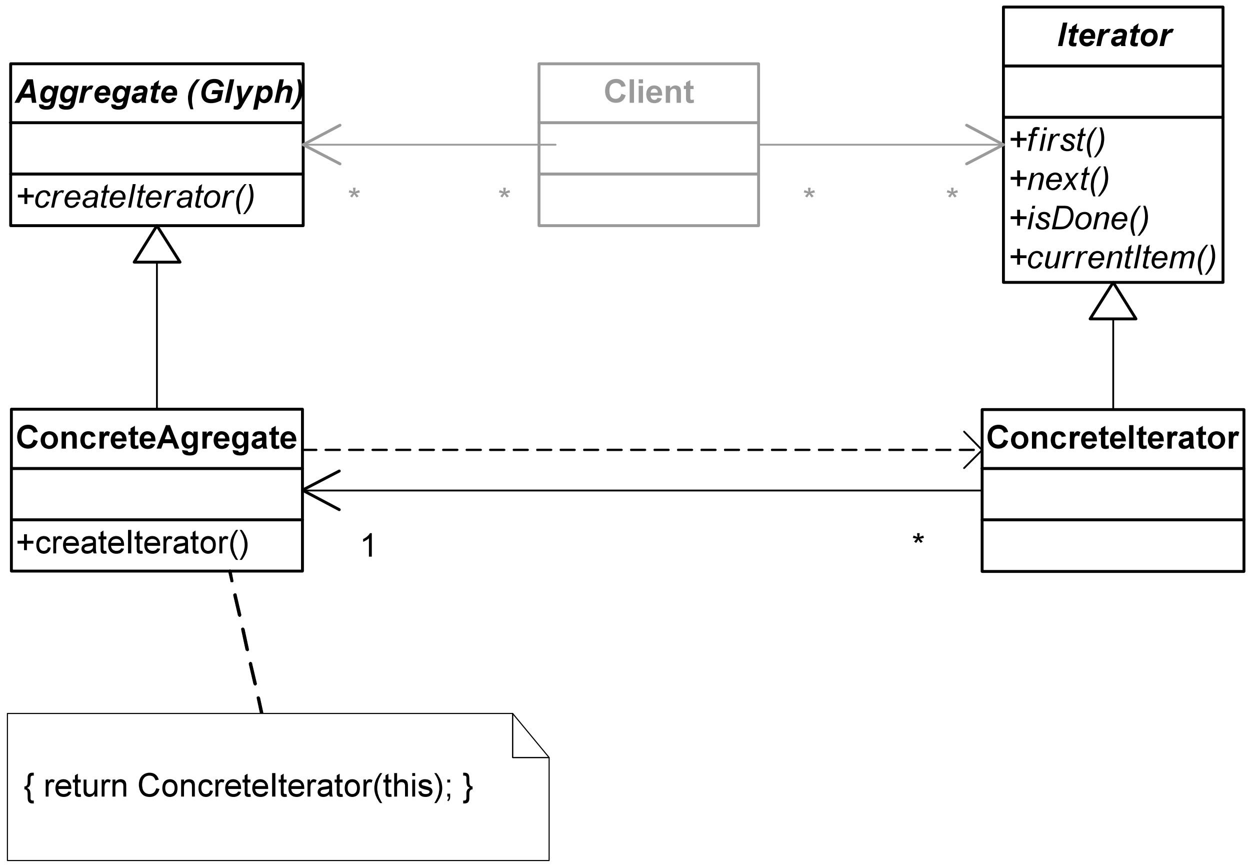
Abstract factory: class diagram



Iterator (Behavioral)

- Intent
  - Access elements of an aggregate sequentially without exposing its representation
- Applicability
  - Require multiple traversal algorithms over an aggregate
  - Require a uniform traversal interface over different aggregates
  - When aggregate classes and traversal algorithm must vary independently

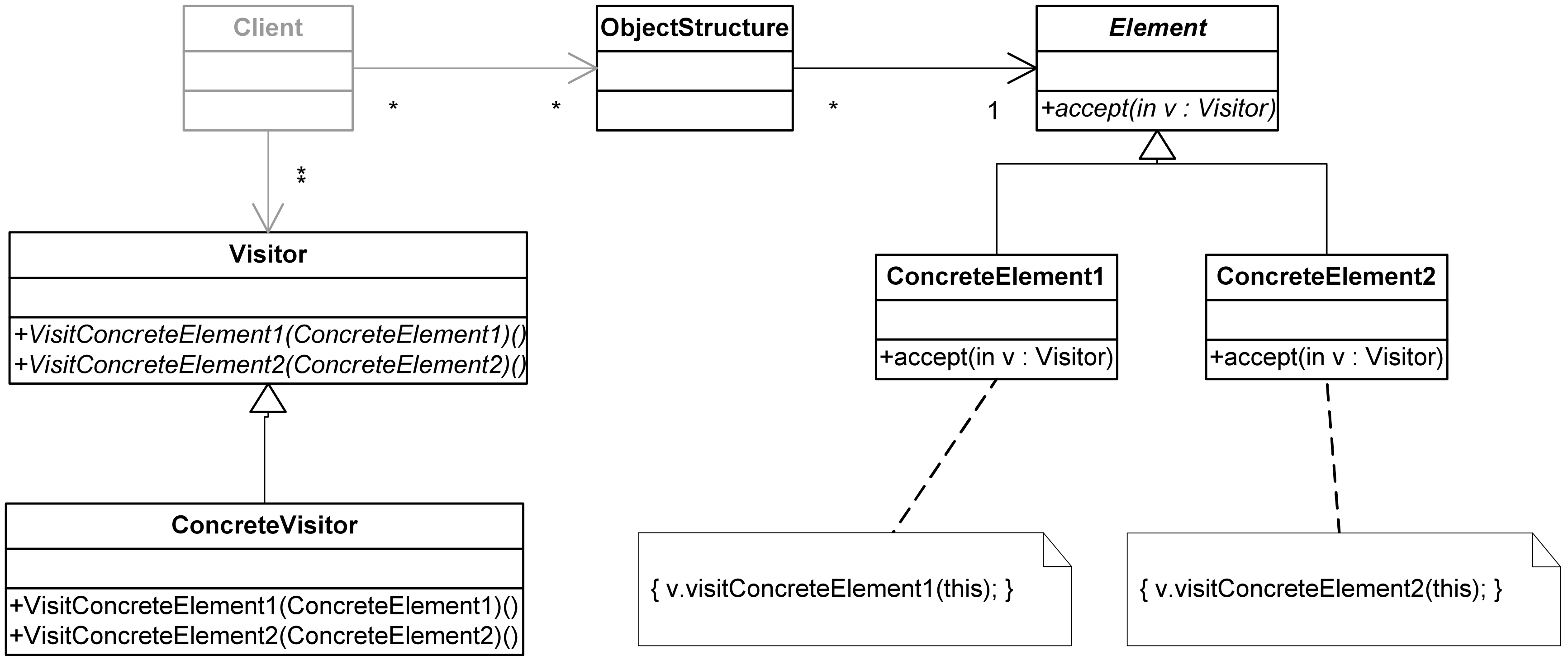
Iterator: class diagram



Visitor (Behavioral)

- Intent
  - Centralize operations on an object structure so that they can vary independently but still behave polymorphically
- Applicability
  - When classes define many unrelated operations
  - Class relationships of objects in the structure rarely change, but the operations on them change often
  - Algorithms over the structure maintain state that's updated during traversal

Visitor: class diagram

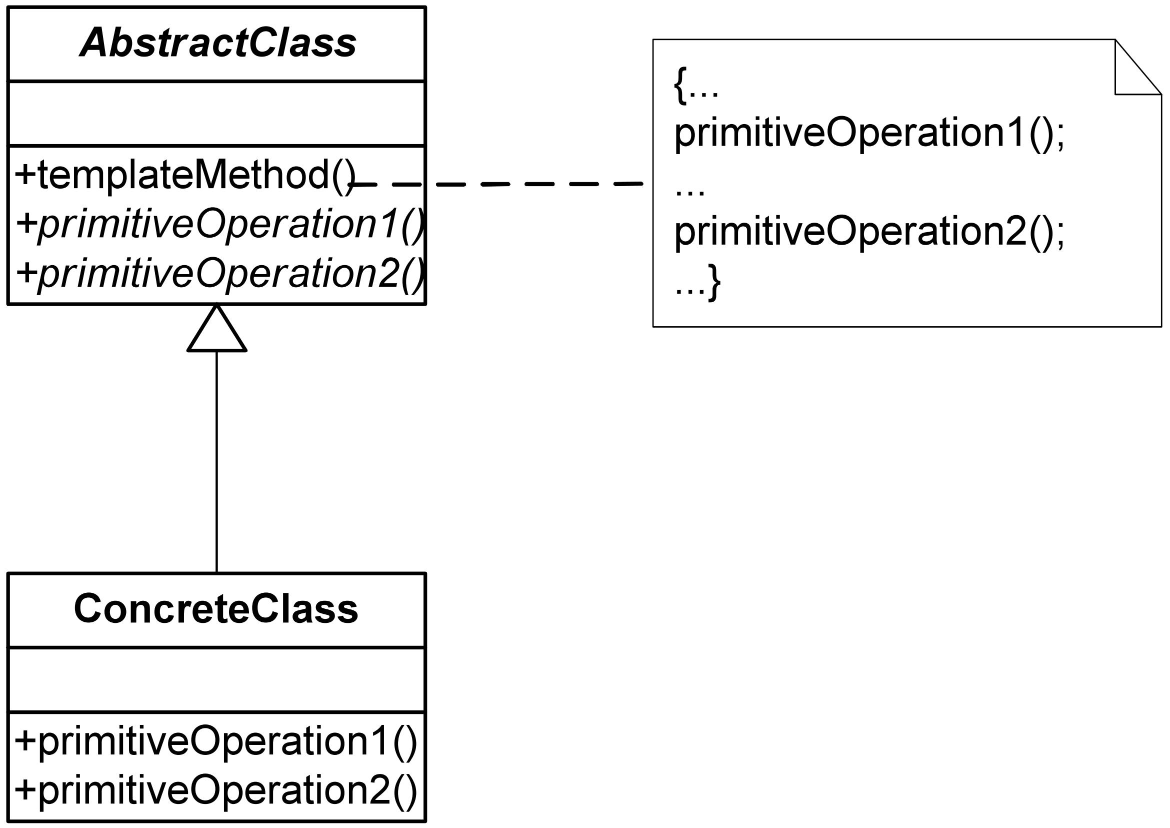


Client
ObjectStructure
Element
+accept(in v : Visitor)
*
*
*
1
*
*
Visitor
+VisitConcreteElement1(ConcreteElement1)()
+VisitConcreteElement2(ConcreteElement2)()
ConcreteVisitor
+VisitConcreteElement1(ConcreteElement1)()
+VisitConcreteElement2(ConcreteElement2)()
ConcreteElement1
+accept(in v : Visitor)
ConcreteElement2
+accept(in v : Visitor)
{ v.visitConcreteElement1(this); }
{ v.visitConcreteElement2(this); }

Template Method (Behavioral)

- Intent
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses
- Applicability
  - To implement invariant aspects of an algorithm once and let subclasses define variant parts
  - To localize common behavior in a class to increase code reuse
  - To control subclass extensions

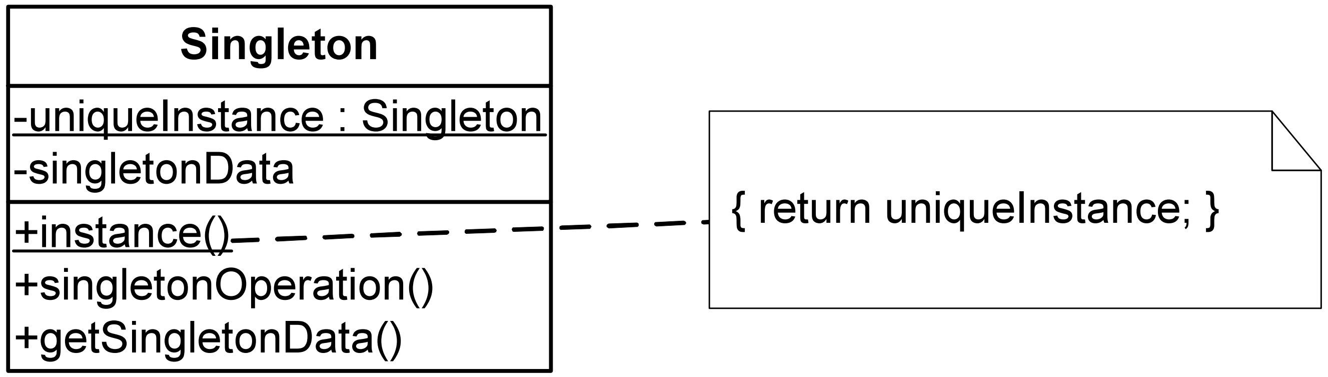
Template Method: class diagram



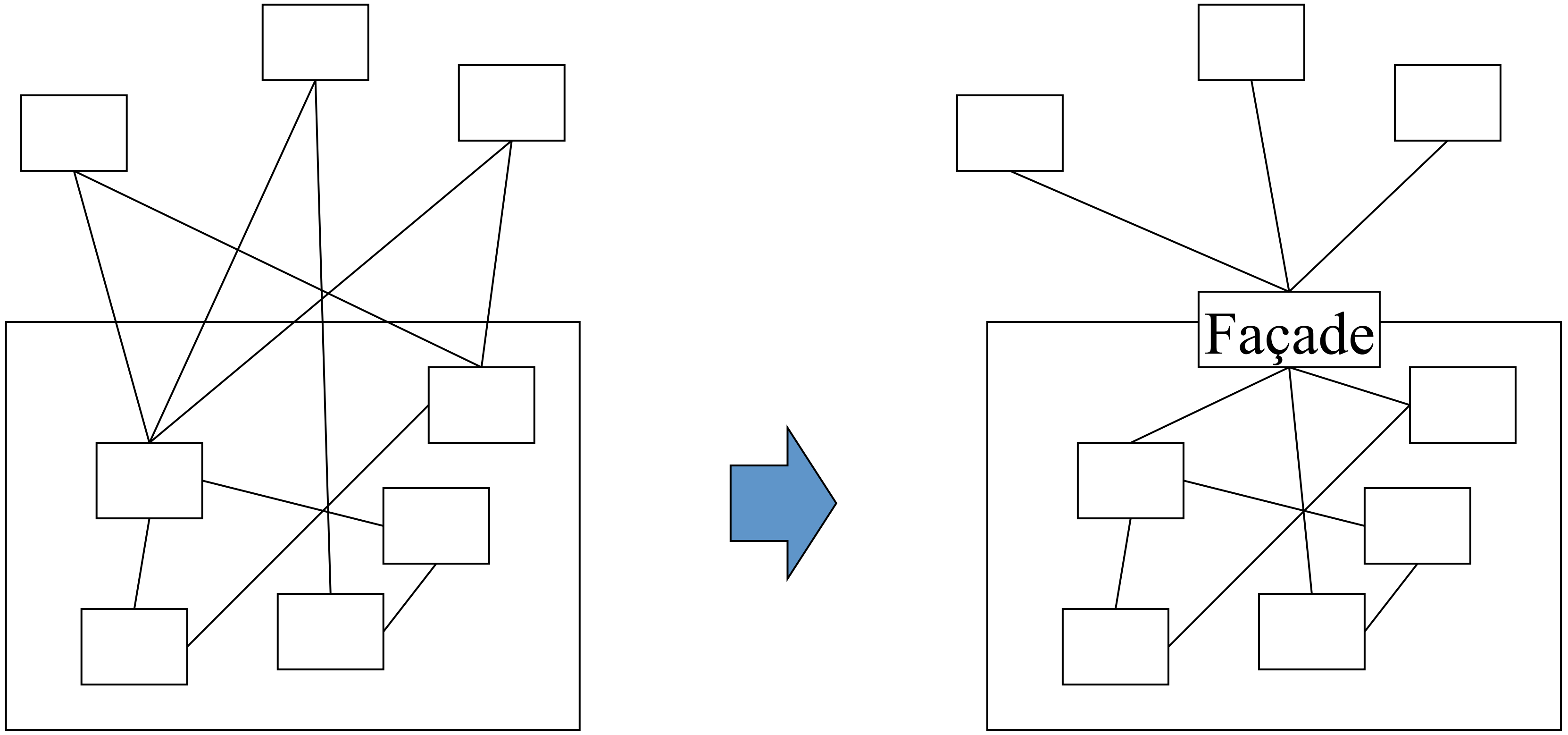
Singleton (Creational)

- Intent
  - Ensure a class only ever has one instance, and provide a global point of access to it
- Applicability
  - When there must be exactly one instance of a class, and it must be accessible from a well-known access point
  - When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code

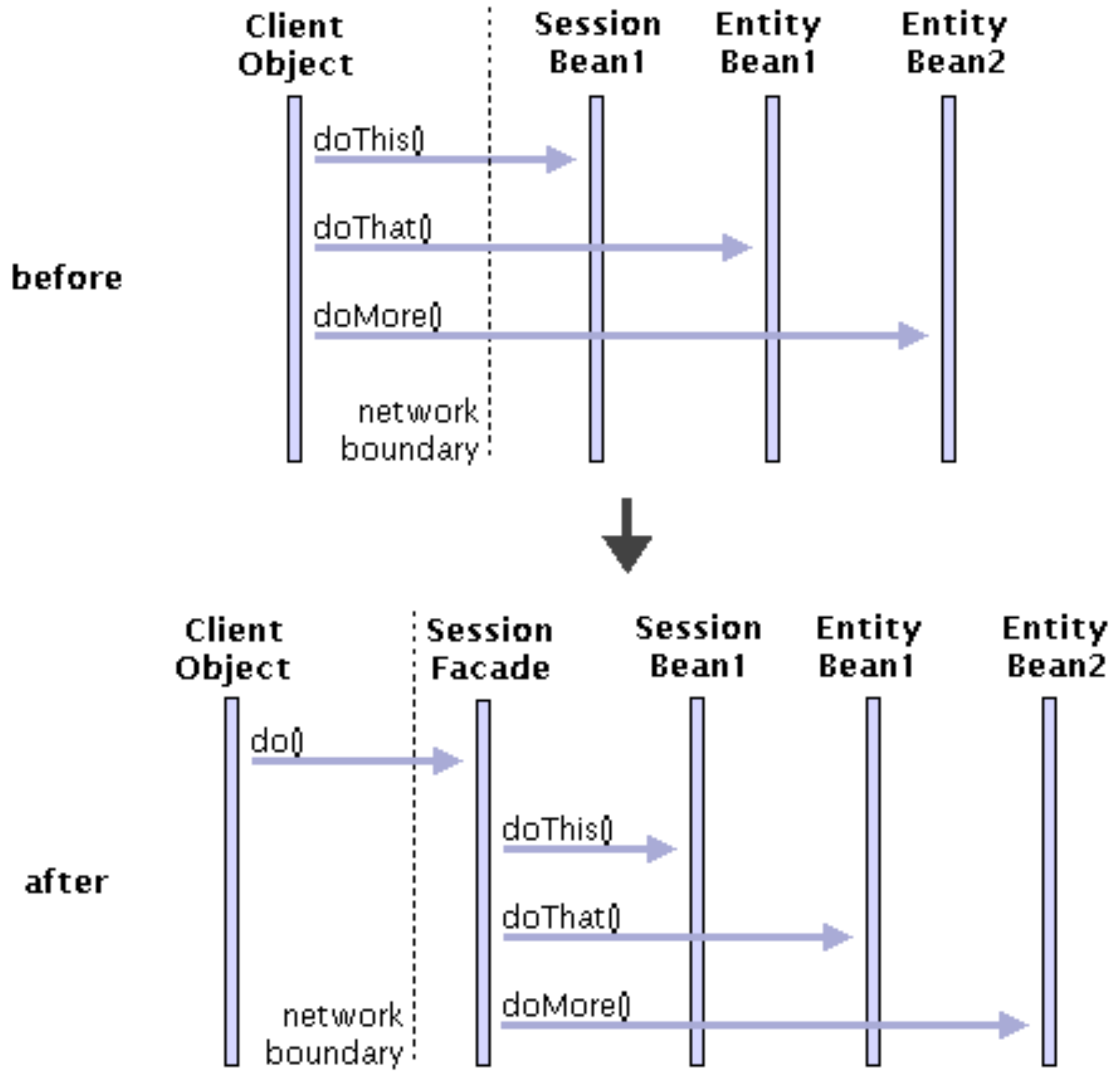
Singleton: class diagram



Façade



Façade

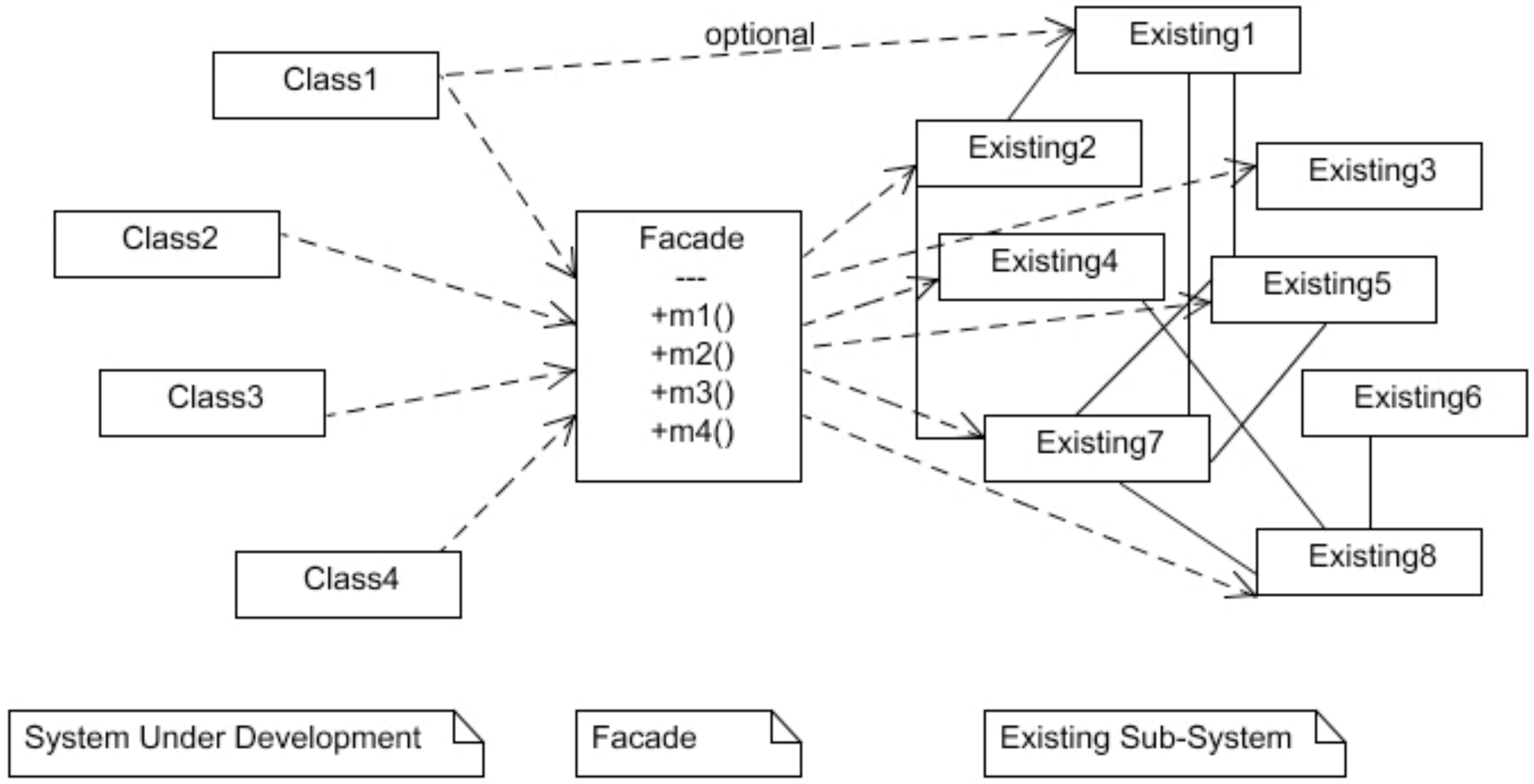


Client Object
Session Bean1
Entity Bean1
Entity Bean2
doThis()
doThat()
doMore()
network boundary
before
Client Object
Session Facade
Session Bean1
Entity Bean1
Entity Bean2
do()
doThis()
doThat()
doMore()
network boundary
after

Façade (Structural)

- Intent
  - Provide a unified interface to a set of interfaces in a subsystem
  - Façade defines a higher-level interface that makes the subsystem easier to use
- Applicability
  - To provide a simple interface to a complex subsystem
  - This interface is good enough for most clients; more sophisticated clients can look beyond the façade
  - To decouple the classes of the subsystem from its clients and other subsystems, thereby promoting subsystem independence and portability

Façade: class diagram



Patterns at Different Levels

- Applicable in most stages of the OO lifecycle
  - Analysis, design, implementation, reviews, documentation, reuse, and refactoring
- Analysis patterns
  - Typical solutions to recuring anlysis problems
  - See Analysis Patterns, Fowler; Addison-Wesley, 1996
- Architectural patterns
  - See POSA
- Design patterns
  - Most GoF design patterns are applicable both at the architectural and detailed design
- Idioms
  - Smalltalk Best Practice Patterns, Beck; Prentice Hall, 1997
  - Concurrent Programming in Java, Lea; Addison-Wesley, 1997
  - Advanced C++, Coplien, Addison-Wesley, 1992
  - Effective C++: 50 Specific Ways to Improve Your Programs and Design (2nd Edition), Scott Meyers, Addison-Wesley, (September 1997)
  - More Effective C++: 35 New Ways to Improve Your Programs and Designs, Scott Meyers, Addison-Wesley (December 1995)

Observations

- Patterns permit design at a more abstract level
  - Treat many class/object interactions as a unit
  - Often beneficial after initial design
  - Targets for class refactorings
- Variation-oriented design
  - Consider what design aspects are variable
  - Identify applicable pattern(s)
  - Vary patterns to evaluate tradeoffs
  - Repeat

MapReduce and Hadoop

based on material by

K. Madurai and B. Ramamurthy

Big-data

- Data mining huge amounts of data collected in a wide range of domains from astronomy to healthcare has become essential for planning and performance
- We are in a knowledge economy
  - Data is an important asset to any organization
  - Discovery of knowledge; Enabling discovery; annotation of data
- We are looking at newer
  - programming models, and
  - Supporting algorithms and data structures

What is MapReduce?

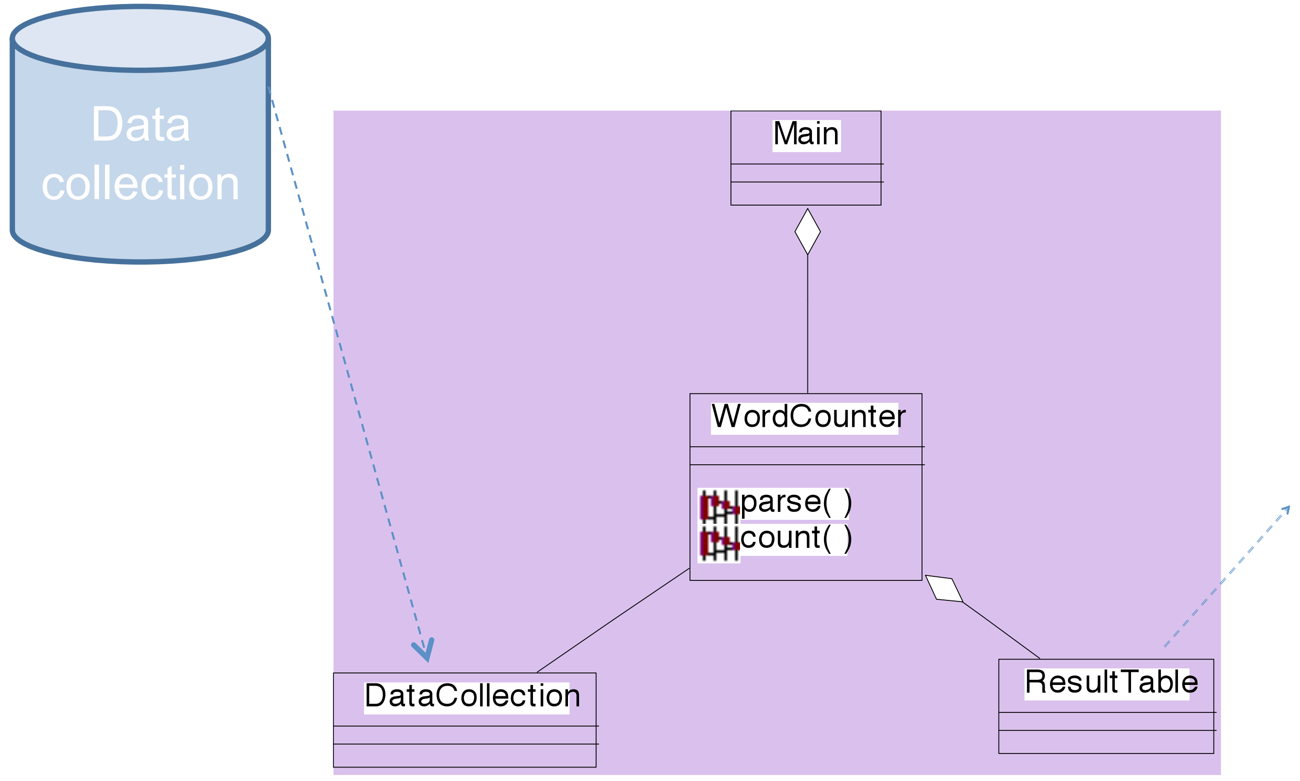
- MapReduce is a programming model
  - Google has used successfully in processing its “big-data” sets (~ 20000 peta bytes per day)
- Users specify the computation in terms of a map and a reduce function
  - Underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, and
  - Underlying system also handles machine failures, efficient communications, and performance issues

Towards MapReduce

- Consider a large data collection:
  - {web, weed, green, sun, moon, land, part, web, green, …}
  - Problem: Count the occurrences of the different words in the collection

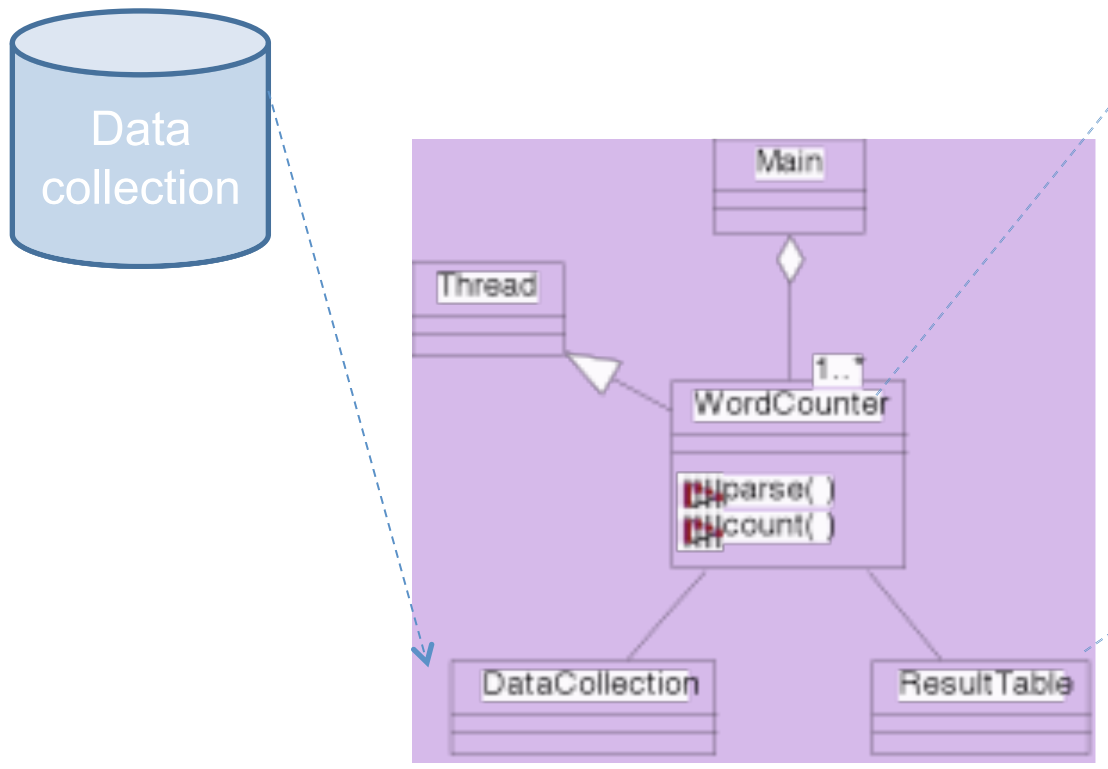
Word Counter and Result Table

{web, weed, green, sun, moon, land, part, web, green,…}



web	**2**
weed	**1**
green	**2**
sun	**1**
moon	**1**
land	**1**
part	**1**

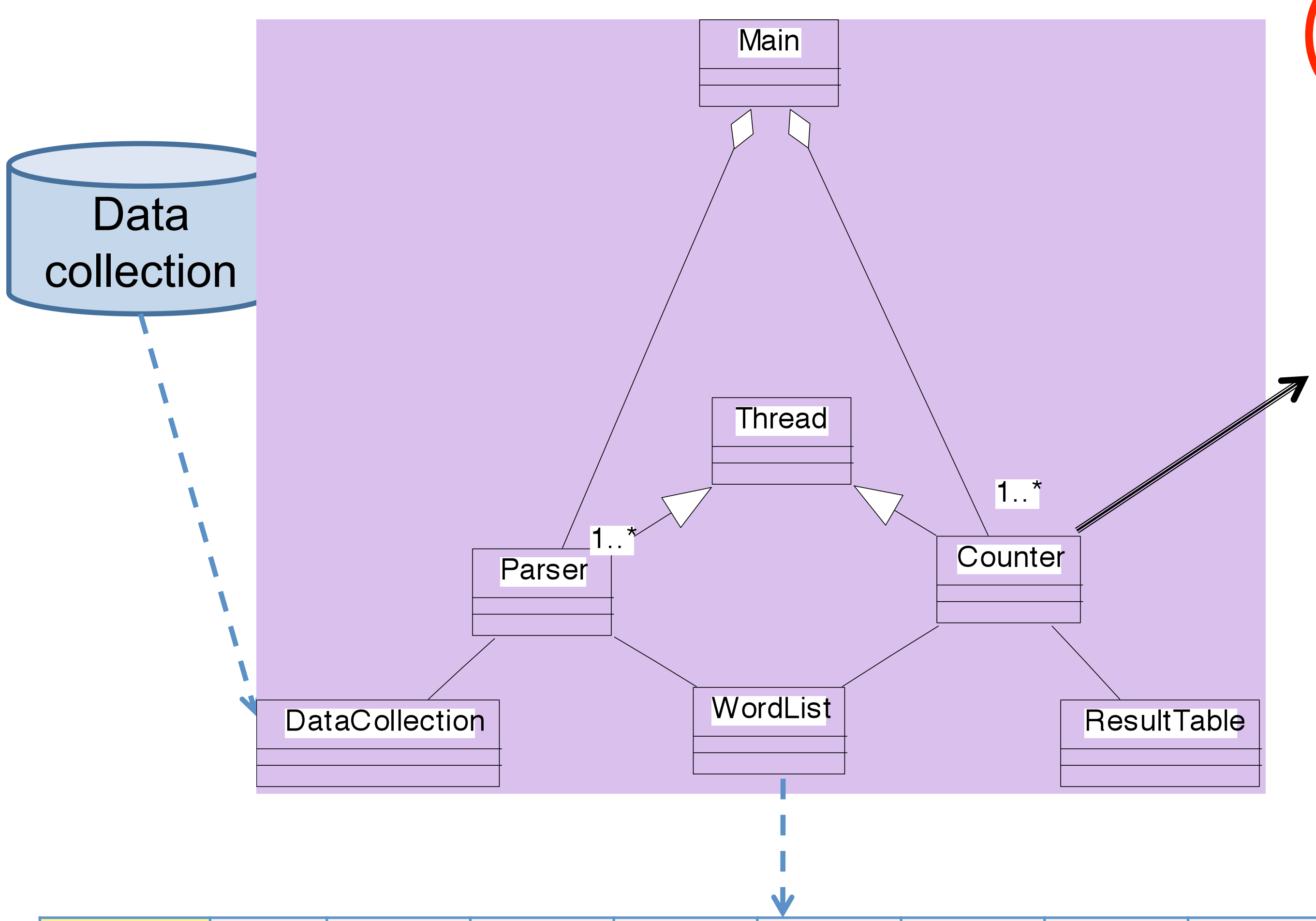
Multiple Instances of Word Counter



web	2
weed	1
green	2
sun	1
moon	1
land	1
part	1

Observe:
Multi-thread
Lock on shared data

Improve Word Counter for Performance



No need for lock

web	2
weed	1
green	2
sun	1
moon	1
land	1
part	1

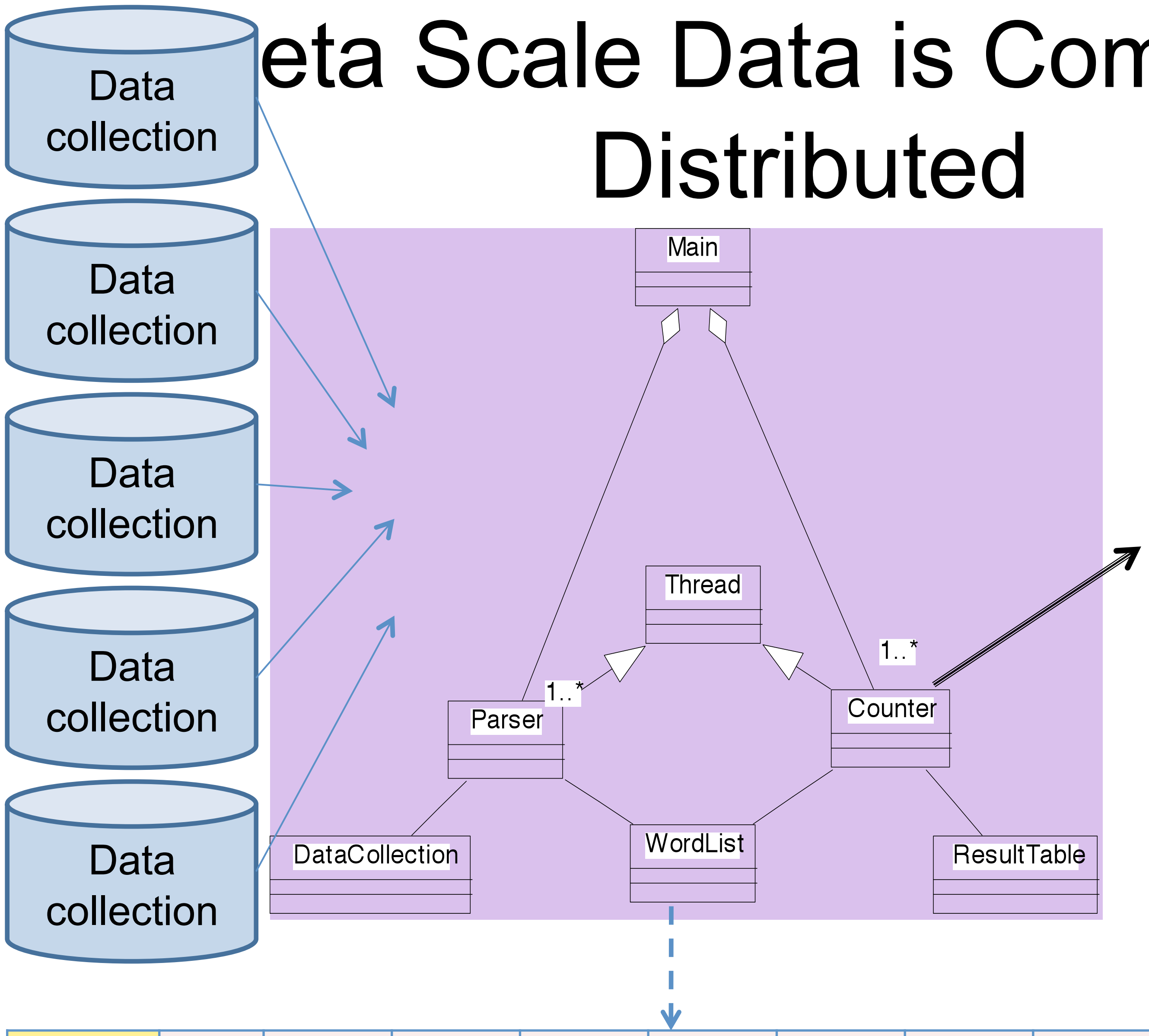
Separate counters

KEY	web	weed	green	sun	moon	land	part	web	green	…….
VALUE										

Addressing the Scale Issue

- Single machine cannot serve all the data: you need a distributed special (file) system
- Failure is norm and not an exception
  - File system has to be fault-tolerant: replication, checksum
- Data transfer bandwidth is critical (location of data)
- Critical aspects: fault tolerance + replication + load balancing, monitoring
- Exploit parallelism afforded by splitting parsing and counting

eta Scale Data is Commonly Distributed

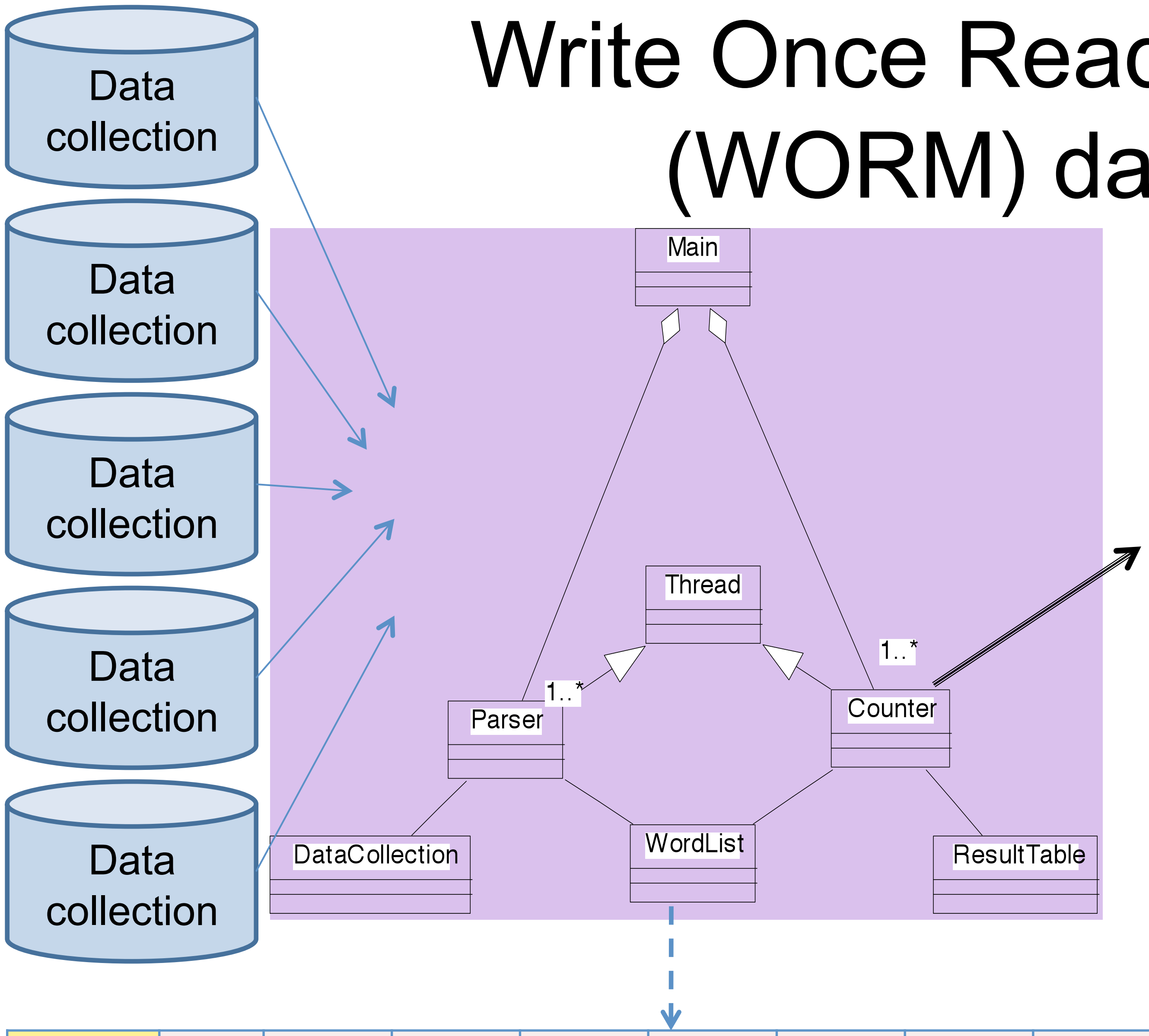


web	**2**
weed	**1**
green	**2**
sun	**1**
moon	**1**
land	**1**
part	**1**

Issue: managing the large scale data

KEY	**web**	**weed**	**green**	**sun**	**moon**	**land**	**part**	**web**	**green**	**…….**
VALUE										

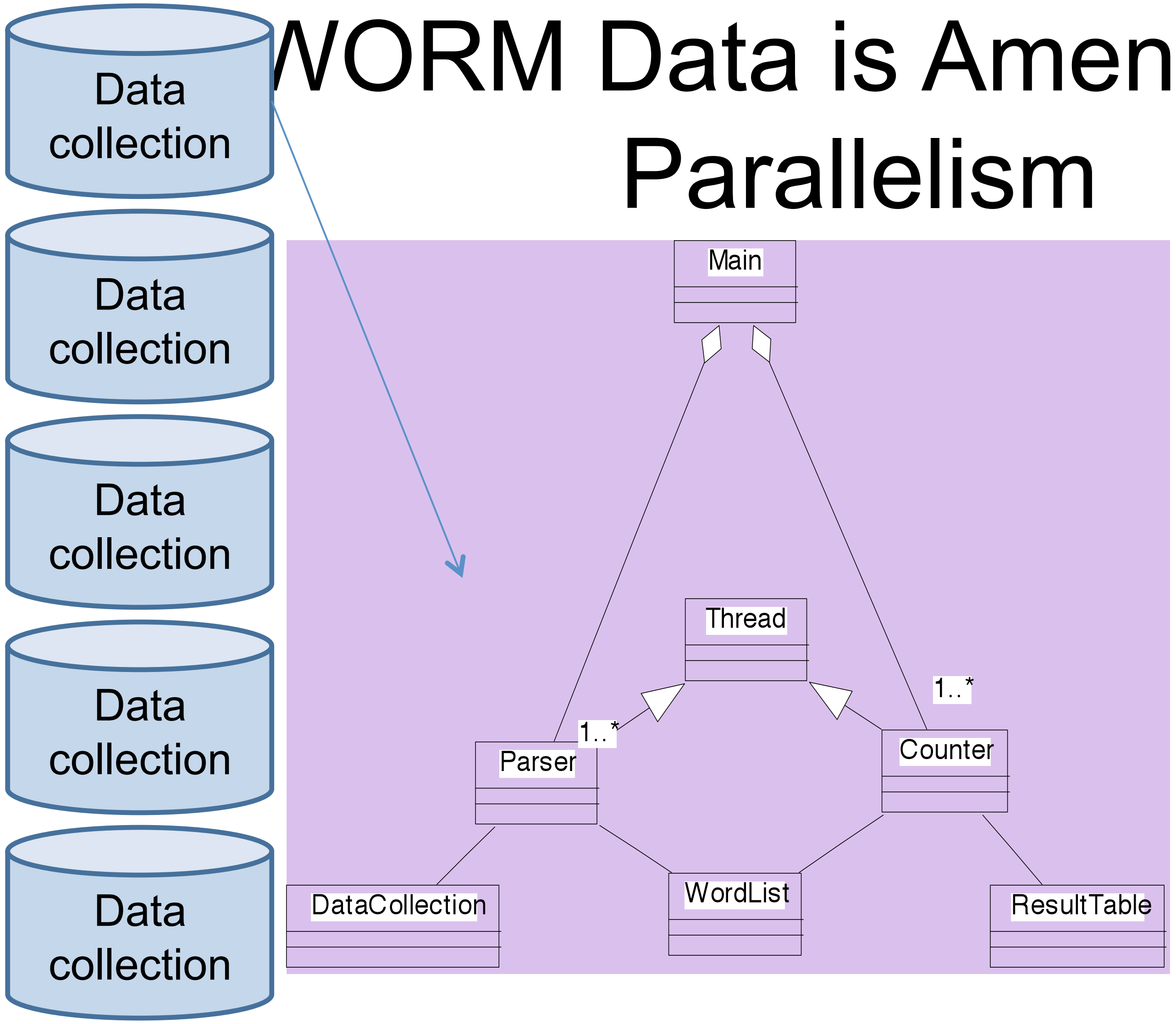
Write Once Read Many (WORM) data



web	2
weed	1
green	2
sun	1
moon	1
land	1
part	1

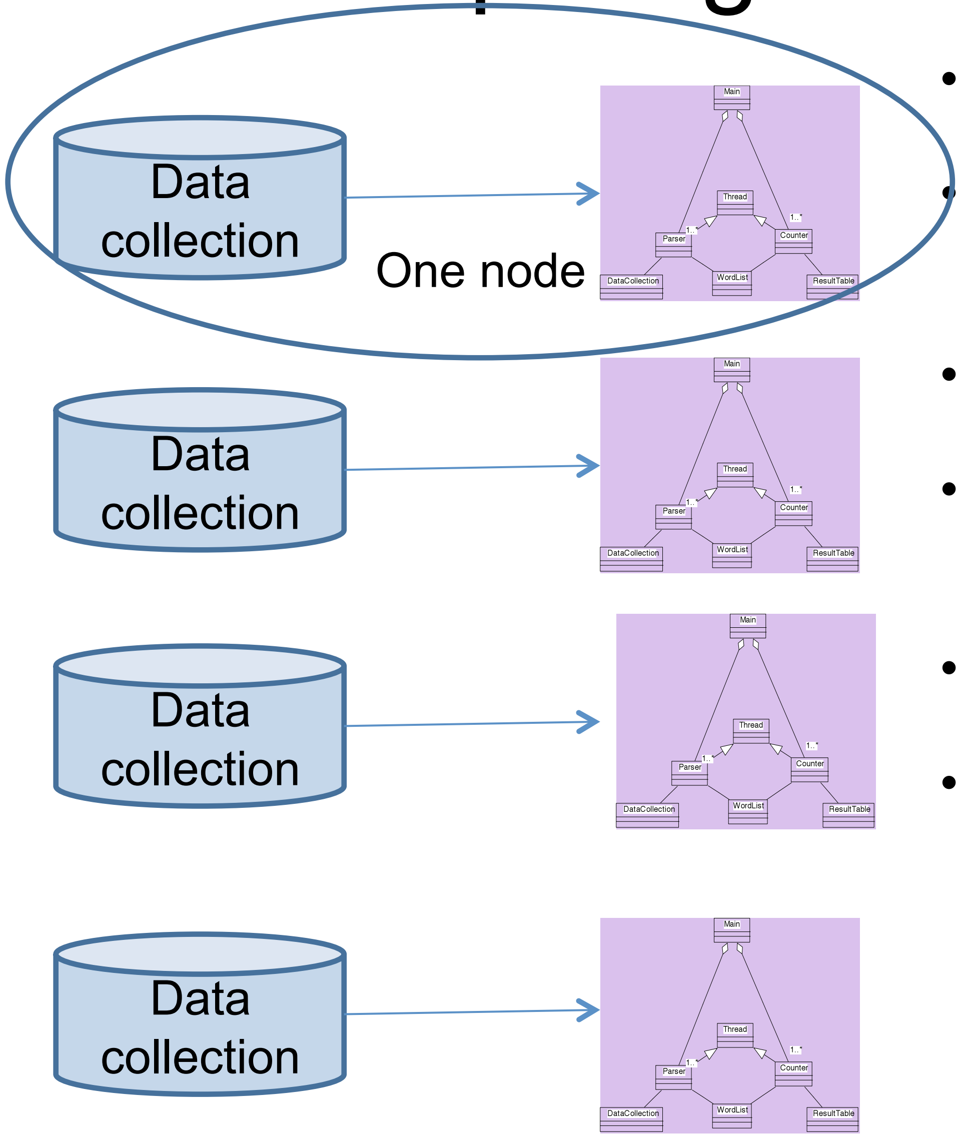
KEY	web	weed	green	sun	moon	land	part	web	green	…….
VALUE										

WORM Data is Amenable to Parallelism



- Data with WORM characteristics : yields to parallel processing
- Data without dependencies: yields to out of order processing

Divide and Conquer: Provision Computing at Data Location

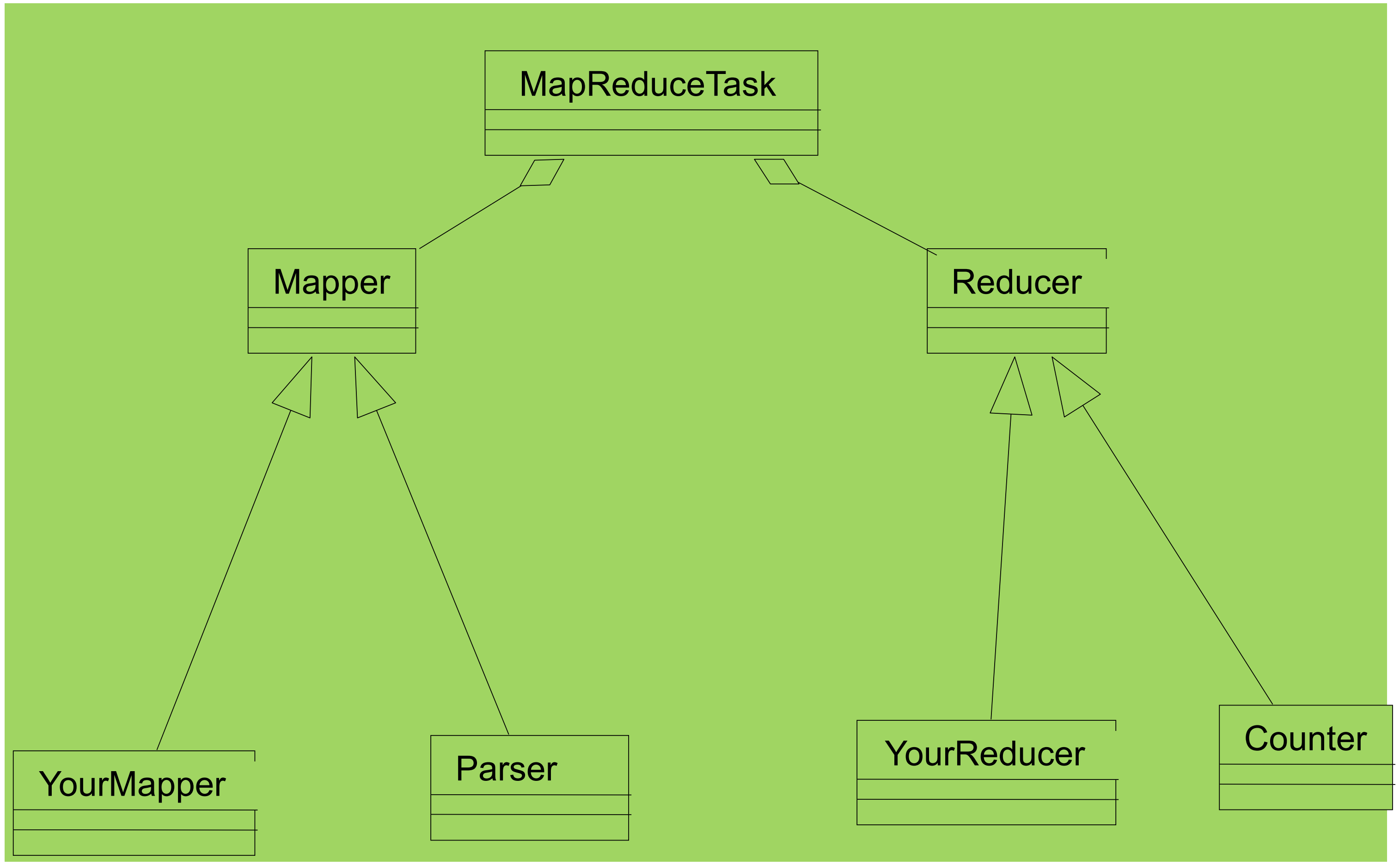


- Our parse is a mapping operation:
- MAP: input → <key, value> pairs

- Our count is a reduceoperation:
- REDUCE: <key, value> pairs reduced

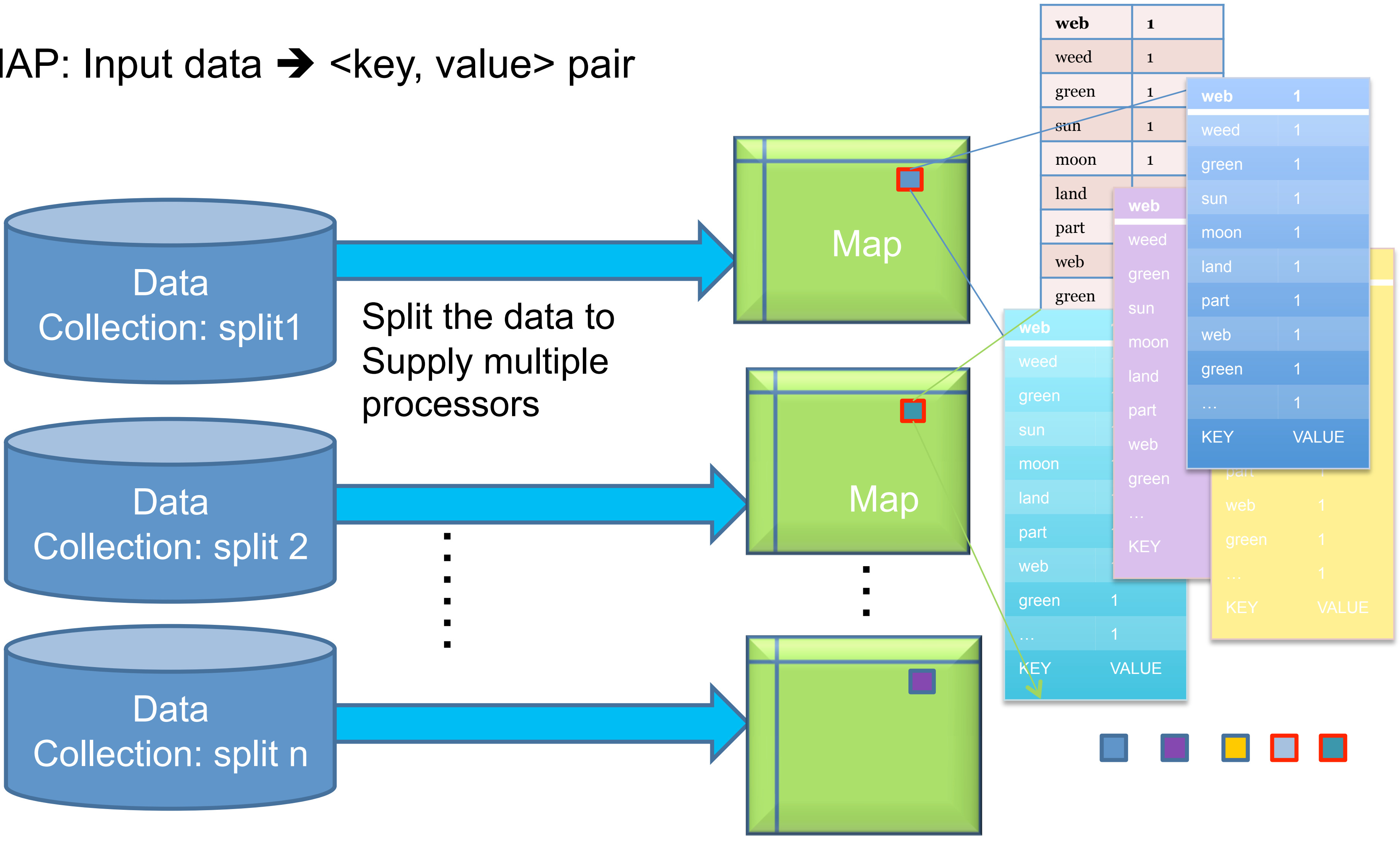
- Map/Reduce originated from Lisp
- But have different meaning here
  - Runtime adds distribution + fault tolerance + replication + monitoring + load balancing to your base application!

Mapper and Reducer



Map Operation

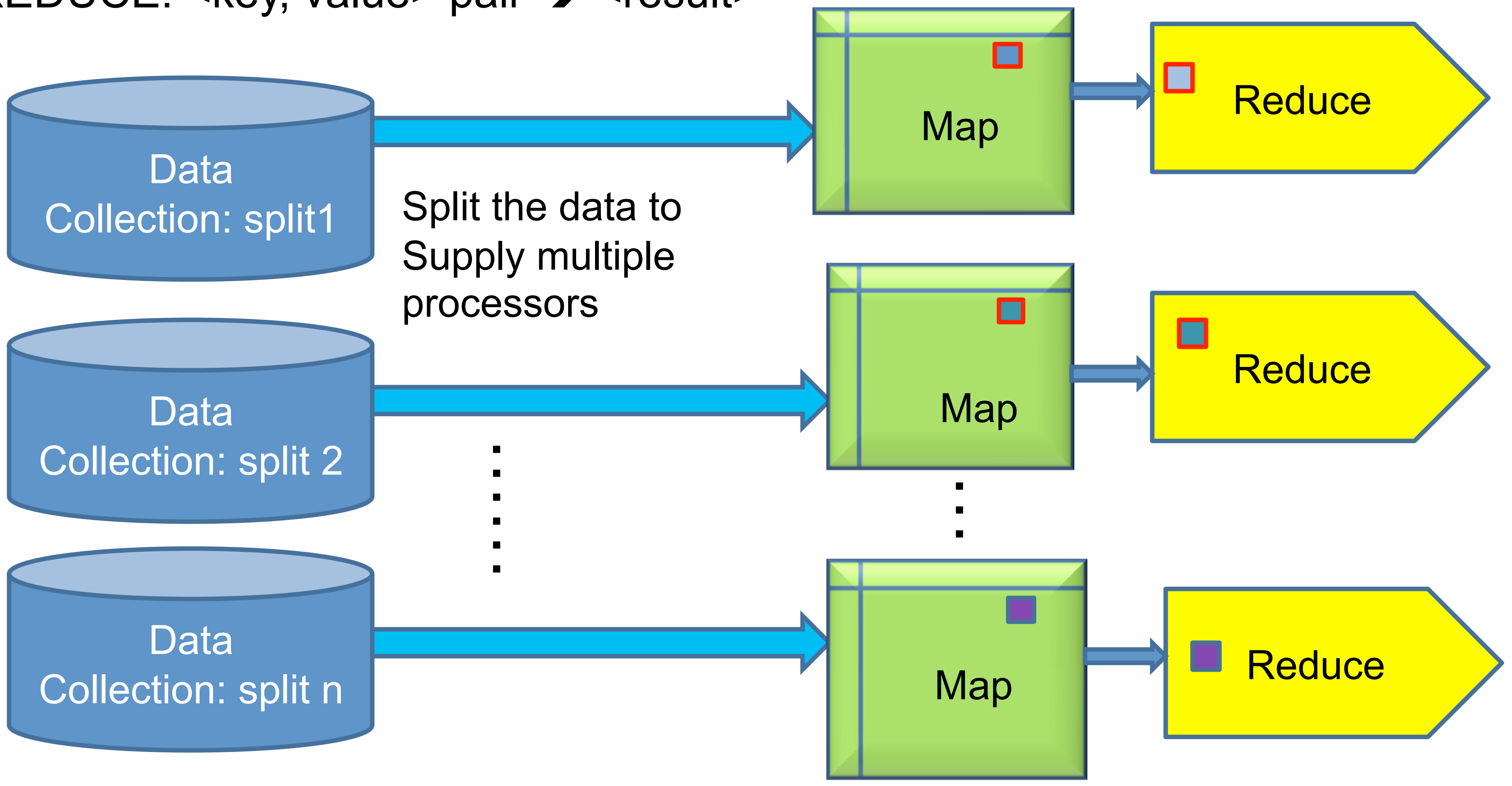
MAP: Input data ➔ <key, value> pair

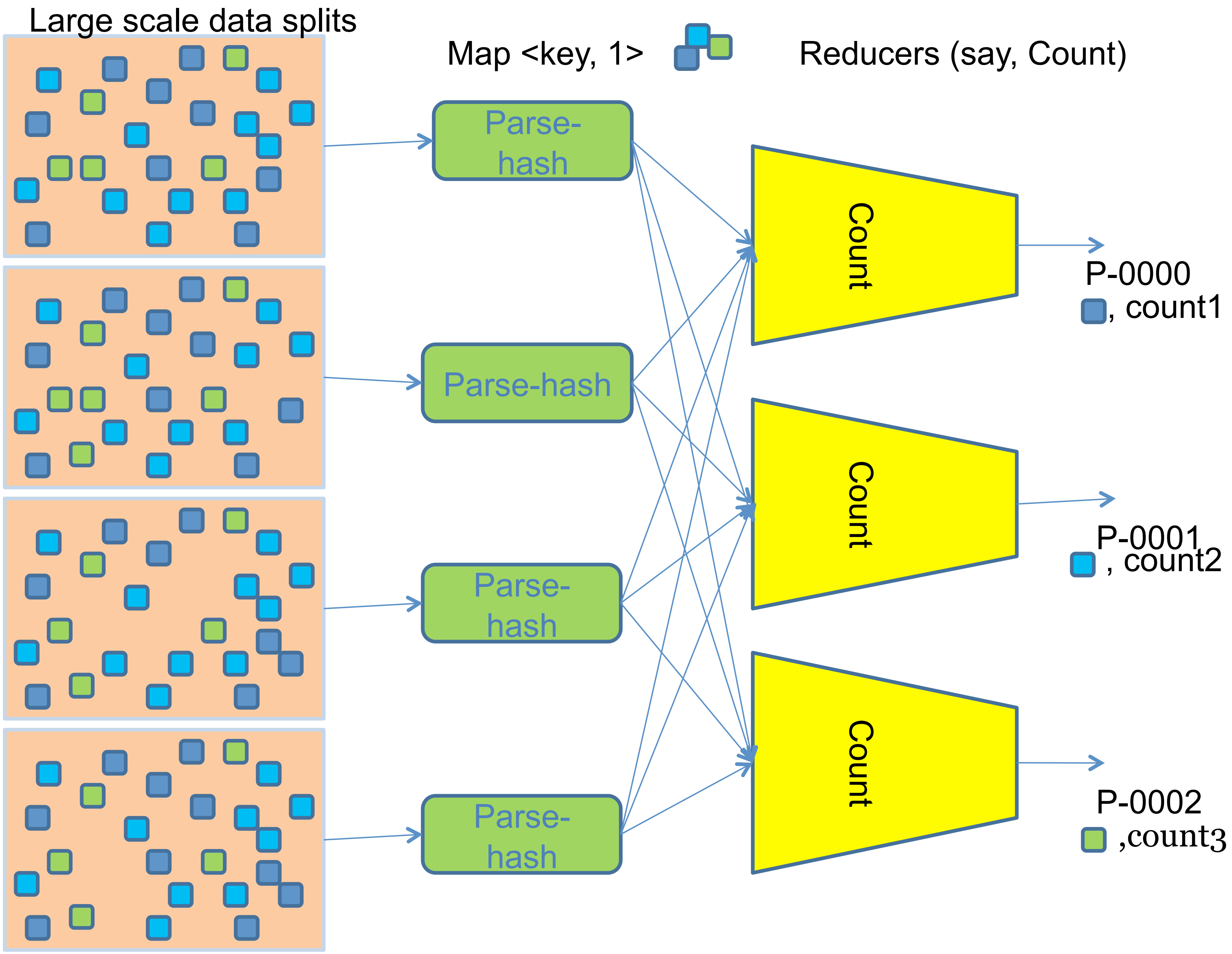


Reduce Operation

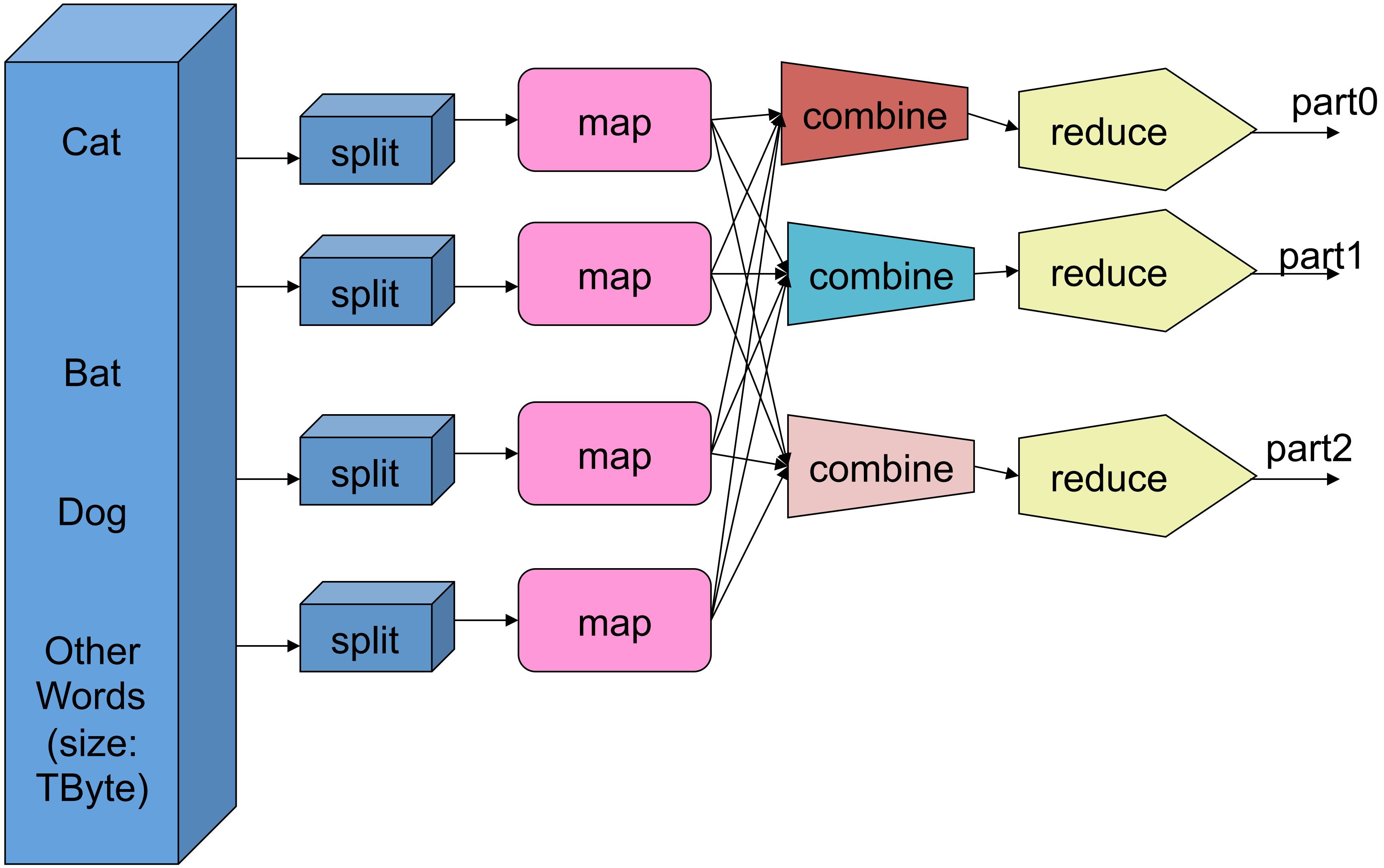
MAP: Input data ➔ <key, value> pair

REDUCE: <key, value> pair ➔ <result>




Large scale data splits
Map <key, 1>
Reducers (say, Count)
Parse-hash
Parse-hash
Parse-hash
Parse-hash
Count
Count
Count
P-0000
, count1
P-0001
, count2
P-0002
,count3

Example


Cat
Bat
Dog
Other Words (size: TByte)
split
split
split
split
map
map
map
map
combine
combine
combine
reduce
reduce
reduce
part0
part1
part2

MapReduce programming model

- Determine if the problem is parallelizable and solvable using MapReduce
- Design and implement solution as Mapper classes and Reducer classes
- Compile the source code with hadoop core
- Package the code as jar executable
- Configure the application (job) as to the number of mappers and reducers (tasks), input and output streams
- Load the data (or use it on previously available data)
- Launch the job and monitor
- Study the result

MapReduce Characteristics

- Very large scale data: peta, exa bytes
- Write once and read many data: allows for parallelism without mutexes
- Map and Reduce are the main operations: simple code
- All the map should be completed before reduce operation starts
- Map and reduce operations are typically performed by the same physical processor
- Number of map tasks and reduce tasks are configurable
- Operations are provisioned near the data
- Commodity hardware and storage
- Runtime takes care of splitting and moving data for operations

“mapreducable” problems

- Google uses it (we think) for wordcount, adwords, pagerank, indexing data
- Simple algorithms such as grep, text-indexing, reverse indexing
- Bayesian classification: data mining domain
- Facebook uses it for various operations: demographics
- Financial services use it for analytics
- Astronomy: Gaussian analysis for locating extra-terrestrial objects
- Expected to play a critical role in semantic web and web3.0

Hadoop

- At Google MapReduce operation are run on a special file system called Google File System (GFS) that is highly optimized for this purpose
- GFS is not open source
- Doug Cutting and Yahoo! reverse engineered the GFS and called it Hadoop Distributed File System (HDFS)
- The software framework that supports HDFS, MapReduce and other related entities is called the project Hadoop or simply Hadoop
- This is open source and distributed by Apache

Basic Features: HDFS

- Highly fault-tolerant
- High throughput
- Suitable for applications with large data sets
- Streaming access to file system data
- Can be built out of commodity hardware

Credits

- Design Patterns, Gamma, et al.; Addison-Wesley, 1995; ISBN 0-201-63361-2; CD version ISBN 0-201-63498-8
- Douglas C. Schmidt