TA Session 10

Ex-6

Running your program

```
java MyFileScript sourceDir commandFile
```

commandFile

```
FILTER
greater_than#1024
ORDER
abs
FILTER
between#2#512
ORDER
size
FILTER
smaller_than#2
ORDER
```

Output

```
big_file.txt
file2.txt
file1.txt
```

SourceDir

Name	Date modified	Type	Size
big_file.txt	17/05/2014 14:57	Text Document	1,690 KB
file1.txt	17/05/2014 14:57	Text Document	310 KB
file2.txt	17/05/2014 15:01	Text Document	4 KB

Parsing

Filters

What would be the Different **Components** of such a System?

Orders

Section

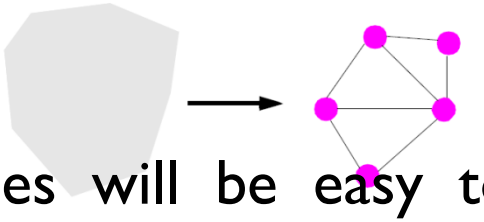
Manager

Different Components

- The exercise definition imposes implementing several different components
  - File parsing, filters/orders, different sections, etc.
- A good design would find a way to divide the task into sub-tasks that are **independent** of one another
- We will build each of these sub-tasks as a different **module**

Modules

- Each of these modules will be independent of one another
  - **Decomposability**



- Each of the modules will be easy to understand without having to know the other modules
  - **Understandability**
- A small change in one module will have a minimal effect on other modules
  - **Modular Continuity**

The different Modules

Parsing Module

- A module that knows the logical order of the *commands file*
  - First comes the *filters*, then the *orders*, …
- Generates the software representation of the program
  - The different sections, different filters and orders, etc.
- This is the only class that knows the logical structure of the file
  - A change in the file structure affects this module only
  - **Modular Continuity**

Sections

- The *commands file* is composed of different sections
- Each section has its own set of filters/orders.
- It thus makes sense to make a module that represents each section
  - Each section **composes** the different filter/order objects
  - Sections are created by the parsing module

Sections cont.

- The section module is independent of the other modules
- It does not know of the *commands file* format
- It does not know of the specific filters/orders
  - Nor does it know their names
  - It works with the general API

The different Modules

Filters Module

- The different filters share a few common features
  - Each filter receives a file and determines whether or not it meets some condition
- It makes sense to put them all in the same module

Filters

Take 1

- Most (all?) filters could be implemented using a few lines of code at most

- Solution 1:
  - Put all filters in the same file
  - Build a small method for each of the filters

```java
protected boolean isFilePassFilter(String name, String value, String value2, File f)
                                                    throws FilterNotFoundException{
    if (name.equals("greater_than")) {
        return isGreaterThan(value,f);
    }else if (name.equals("between")) {
        return isBetween(value,value2,f);
    } else if (name.equals("smaller_than")) {
        return isSmallerThan(value,f);
    }

    throw new FilterNotFoundException(name);
}

public boolean isGreaterThan(String value, File f){
    return false;
}
public boolean isBetween(String value, String value2, File f){
    return false;
}
public boolean isSmallerThan(String value, File f){
    return false;
}
```

Filters

Take 1 – Pros & Cons

- Pros:
  - Compact
  - Requires a single file only
- Cons:
  - Adding a new filter requires modifying a working file
    - Breaks the **open/closed** principle
  - Future filters might be more complex and require more than a few lines of code
    - Single file will become large and hard to maintain

Filters

Take 2

- Implement each filter in its own class
  - Adding a new filter requires modifying only 1-2 classes
  - **Open/closed principle**
- Create a hierarchy of filters
  - Filters that share a functionality can have a common parent
    - size filters, etc.
- Super filter is an **interface**

```java
public boolean isFilePassFilter(Filter filter, File f) {
    return filter.isPass(f);
}


public class GreaterThanFilter implements Filter {
    public boolean isPass(File f) {
        return false;
    }
}
public class SmallerThanFilter implements Filter {
    public boolean isPass(File f) {
        return false;
    }
}
public class BetweenFilter implements Filter {
    public boolean isPass(File f) {
        return false;
    }
}
```

```java
public boolean isFilePassFilter(Filter filter, File f) {
    return filter.isPass(f);
}


public class GreaterThanFilter implements Filter {
    public boolean isPass(File f) {
        return false;
    }
}
public class SmallerThanFilter implements Filter {
    public boolean isPass(File f) {
        return false;
    }
}
public class BetweenFilter implements Filter {
    public boolean isPass(File f) {
        return false;
    }
}
```

Filters

Negation Filter

- Applying the NOT word on a filter creates a negative version of this filter
- This is not a filter on its own, but some additional functionality to an existing filter
  - This functionality can be added to any filter
- What is a good solution for this problem?

Filters

Negation Filter

- Applying the NOT word on a filter creates a negative version of this filter
- This is not a filter on its own, but some additional functionality to an existing filter
  - This functionality can be added to any filter
- What is a good solution for this problem?

Decorator!

The different Modules

Orders Module

- Similar idea to the filter case
  - Use one class per order

Filters/Orders Creation

- Part of the parsing process is to create the different filter/order objects based on the data in the *commands file*
  - greater_than%1024 should result in creating a GreaterThanFilter object

Filters/Orders Creation
Take 1

- Make it part of the **parsing** module
- Pros:
  - Analyzing the different strings in the *commands file* is part of the parsing task
- Cons:
  - The exhaustive list of filters/orders is **independent** of the file format
    - We can add a new filter/order without changing the file format and vice-versa

Filters/Orders Creation

Take 2

- Build a **factory** for each set of classes
  - Filters factory, and Orders factory
  - Each factory is responsible for creating instances of the different classes
- Make each factory part of the module it creates

Factory

- A **creational** design pattern

```
public class FilterFactory {
   public static Filter createFilter(String filterString) {
      if (filterString.equals("greater_than")) {
                  return new GreaterThanFilter();
     }else if(filterString.equals("smaller_than"))
                  return new SmallerThanFilter();
     }…
   }
}
```

Filters/Orders Creation
Factory

- Pros:
  - Adding a new filter/order is confined to a single module
    - Requires adding a new class and modifying the factory class
    - The single choice principle
    - Modular continuity principle
  - The parsing module is focused on parsing the *commands file*
    - It doesn't need to know what possible filters/orders exist

Factory

- Important: you should **not** put all factories in the same module
  - Although they share the same design (Factory pattern), they do not share the same task, and are completely **independent** of one another
- Put factories in the same package as the objects they are generating

Error Handling

- The exceptions mechanism is useful for error handling
- We can define a new exception for each type of error
- Exceptions should be built in a hierarchical structure
  - E.g., all *parsing errors* should have a common ancestor
- Defining a hierarchy of exceptions also allows us to treat different exceptions in the same way, if necessary

Error Handling 2

- Exceptions are an inherent part of the problems they are built for
- As a result, exceptions should be found in the same **module** as the classes that throw them
  - E.g., **filter** exceptions should be found in the **filters** module
- Exceptions that are shared by several modules can reside in the main module

Manager

- The module that runs it all
  - Call the parsing module to parse the file
  - Iterate the different sections
    - Print warnings
    - In each section, traverse files in the *source directory*, filter them, print in the relevant order
  - Etc.

Parsing

Parsing

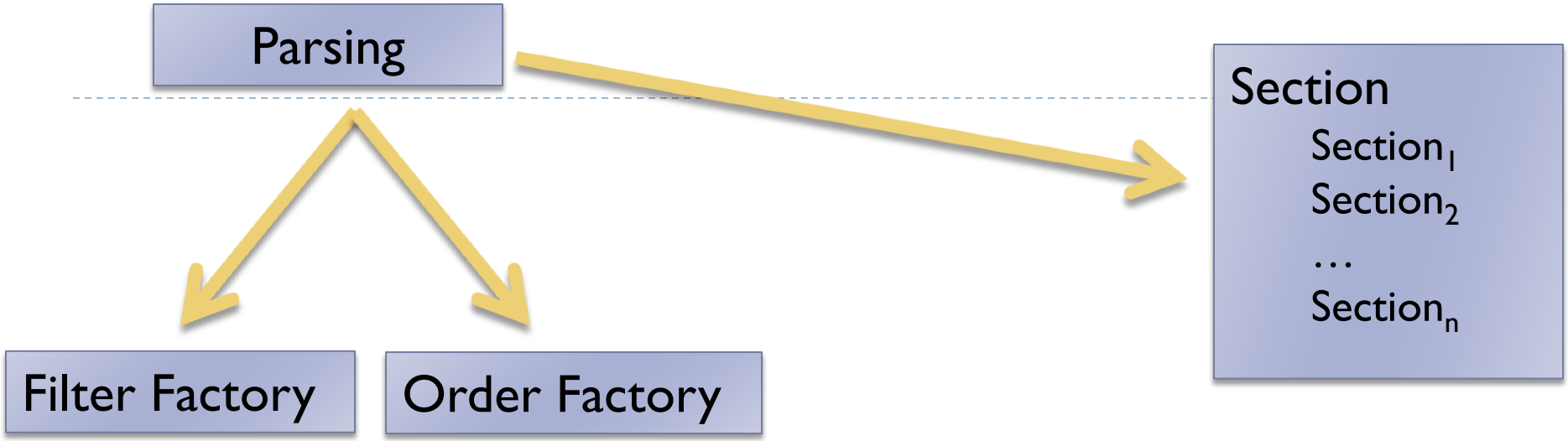
Section

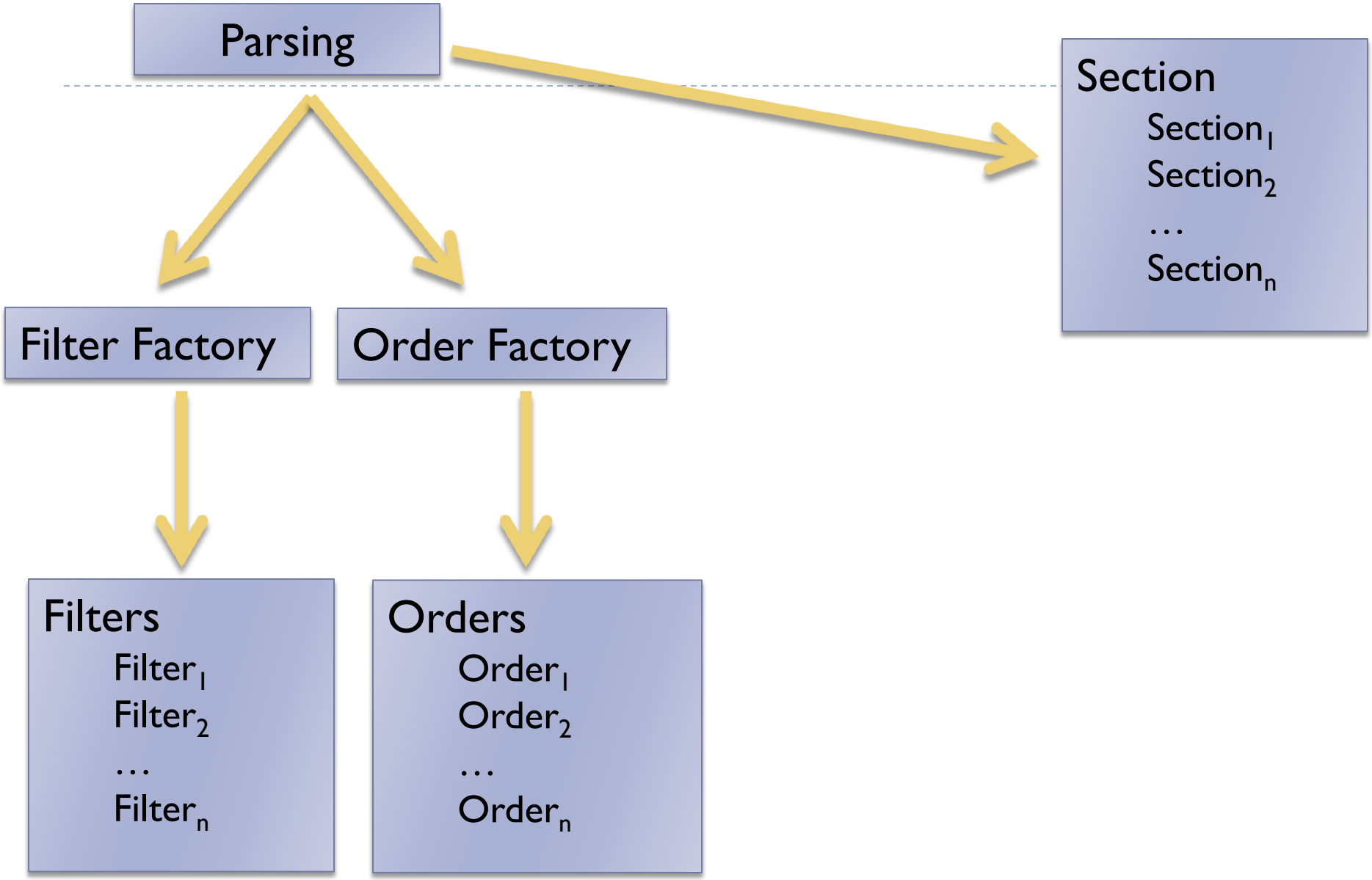
$Section_1$

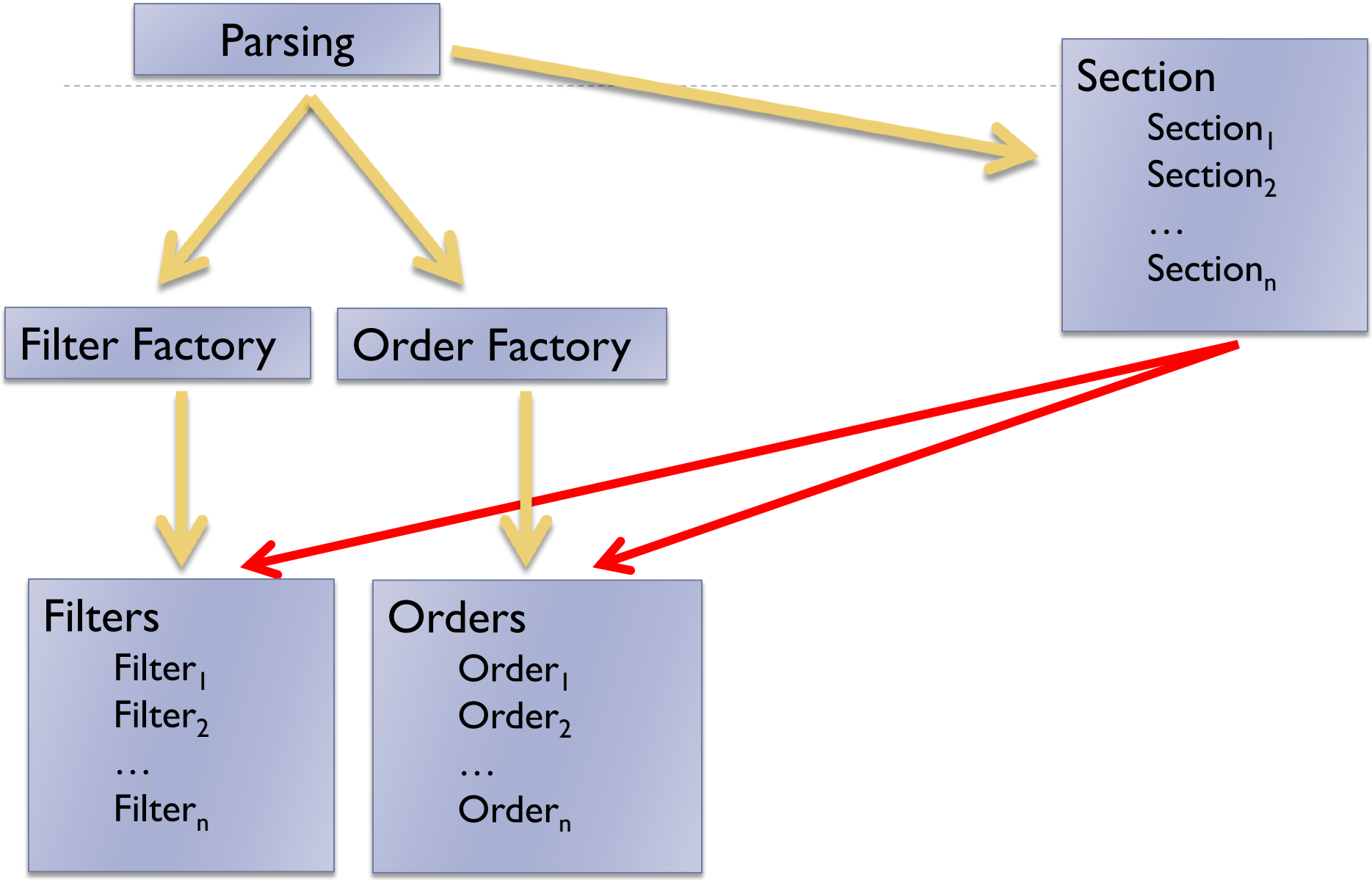
$Section_2$

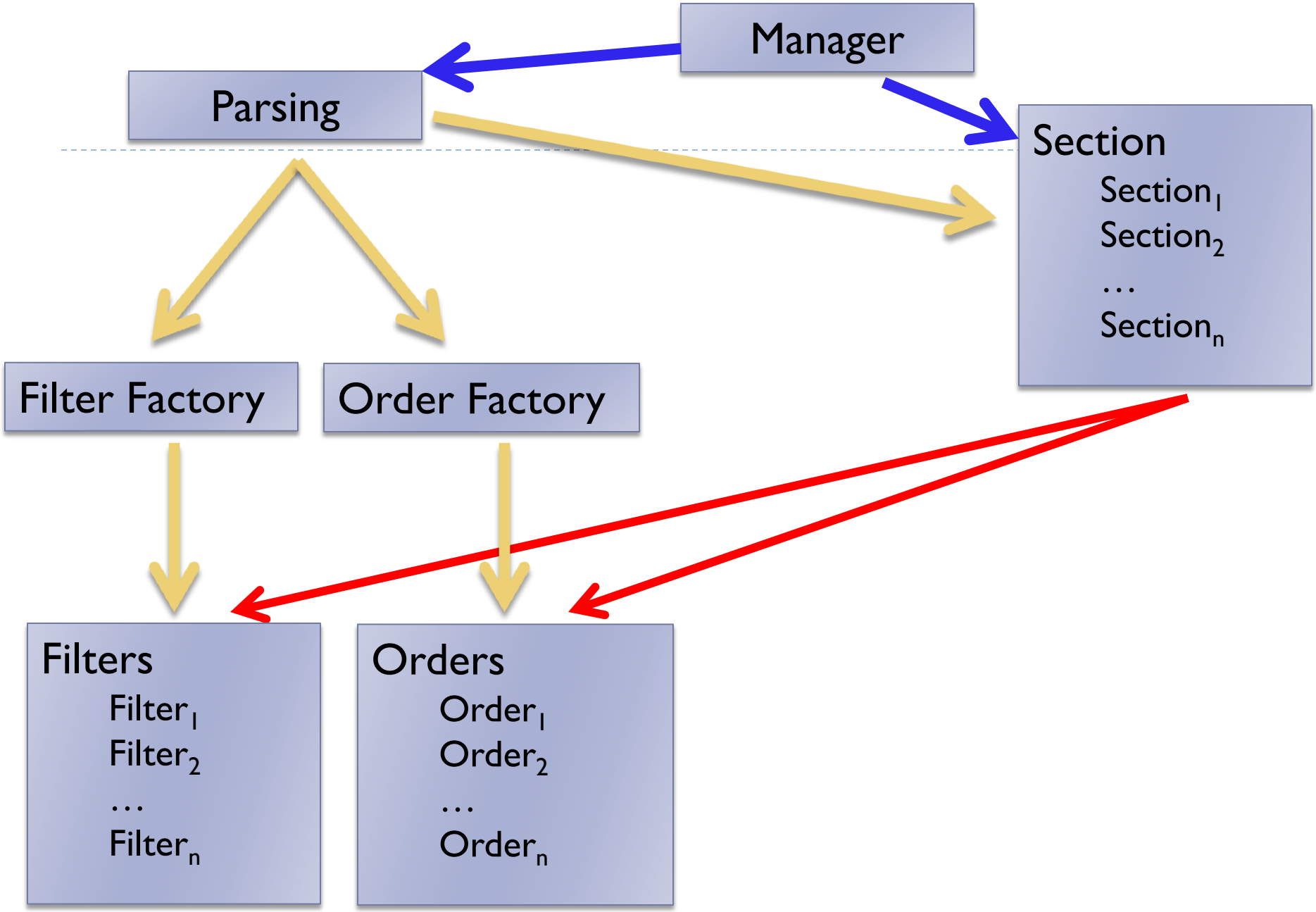
…

$Section_n$


Parsing
Section
Section1
Section2
…
Sectionn
Filter Factory
Order Factory


Parsing
Section
Section1
Section2
…
Sectionn
Filter Factory
Order Factory
Filters
Filter1
Filter2
…
Filtern
Orders
Order1
Order2
…
Ordern


Parsing
Section
Section1
Section2
…
Sectionn
Filter Factory
Order Factory
Filters
Filter1
Filter2
…
Filtern
Orders
Order1
Order2
…
Ordern


Manager
Parsing
Section
Section1
Section2
…
Sectionn
Filter Factory
Order Factory
Filters
Filter1
Filter2
…
Filtern
Orders
Order1
Order2
…
Ordern