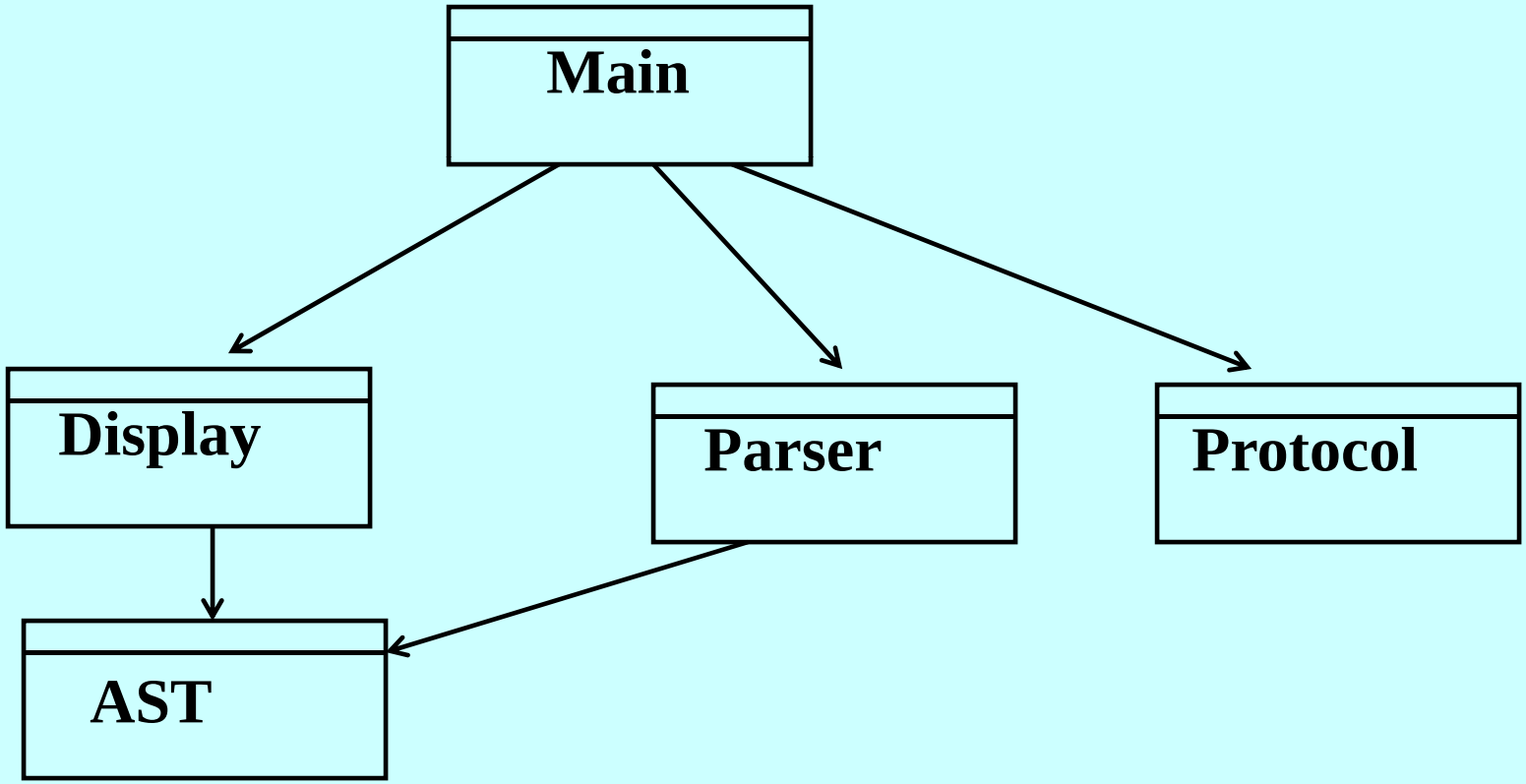
Dependency Relationships

- **uses**
- **depends**

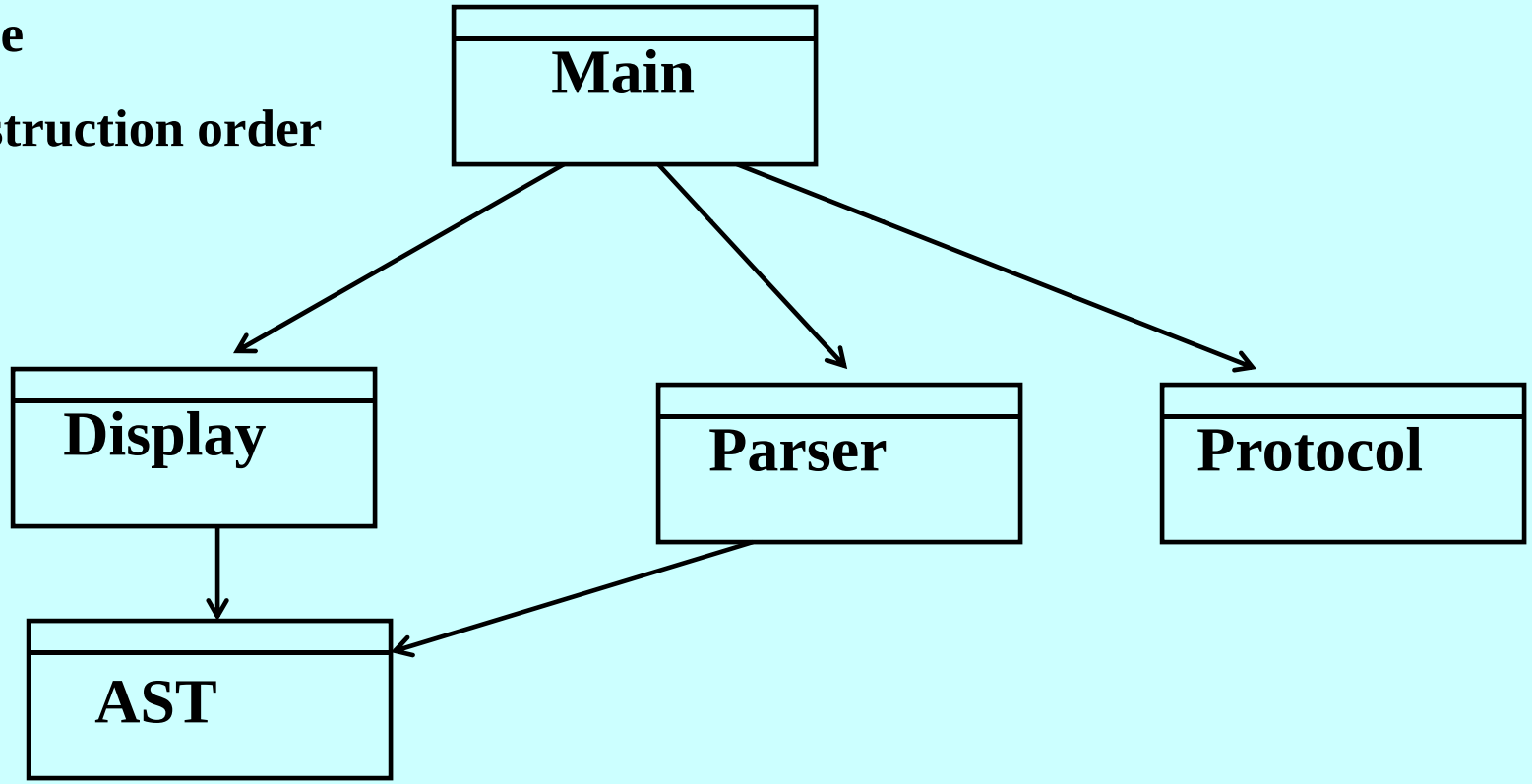
Uses Relationship

- **Part A uses Part B if A refers to B in such a way that the meaning of A depends on the meaning of B.**



Uses Diagram

- **What can we do with a Uses diagram:**
  - **Reasoning**
  - **Reuse**
  - **Construction order**



Uses Diagram

- **Uses Diagram helps us in evaluating the quality of design**

- **Problems:**
  - **Most of the analyses involve finding all parts reachable or reaching a part**

- **Better: if reasoning about a part required looking at only the part it refers to and not others.**

Solution: notion of dependence that stops after one step. To reason about part A, we need to consider only those parts that it depends on.

Specifications

Solution: notion of dependence that stops after one step.

• to reason about part A, we need to consider only those parts that it depends on.

• it is necessary for every part that *A* depends on to be complete, in the sense that its description completely characterizes its behaviour. Such a description is called a *specification*.

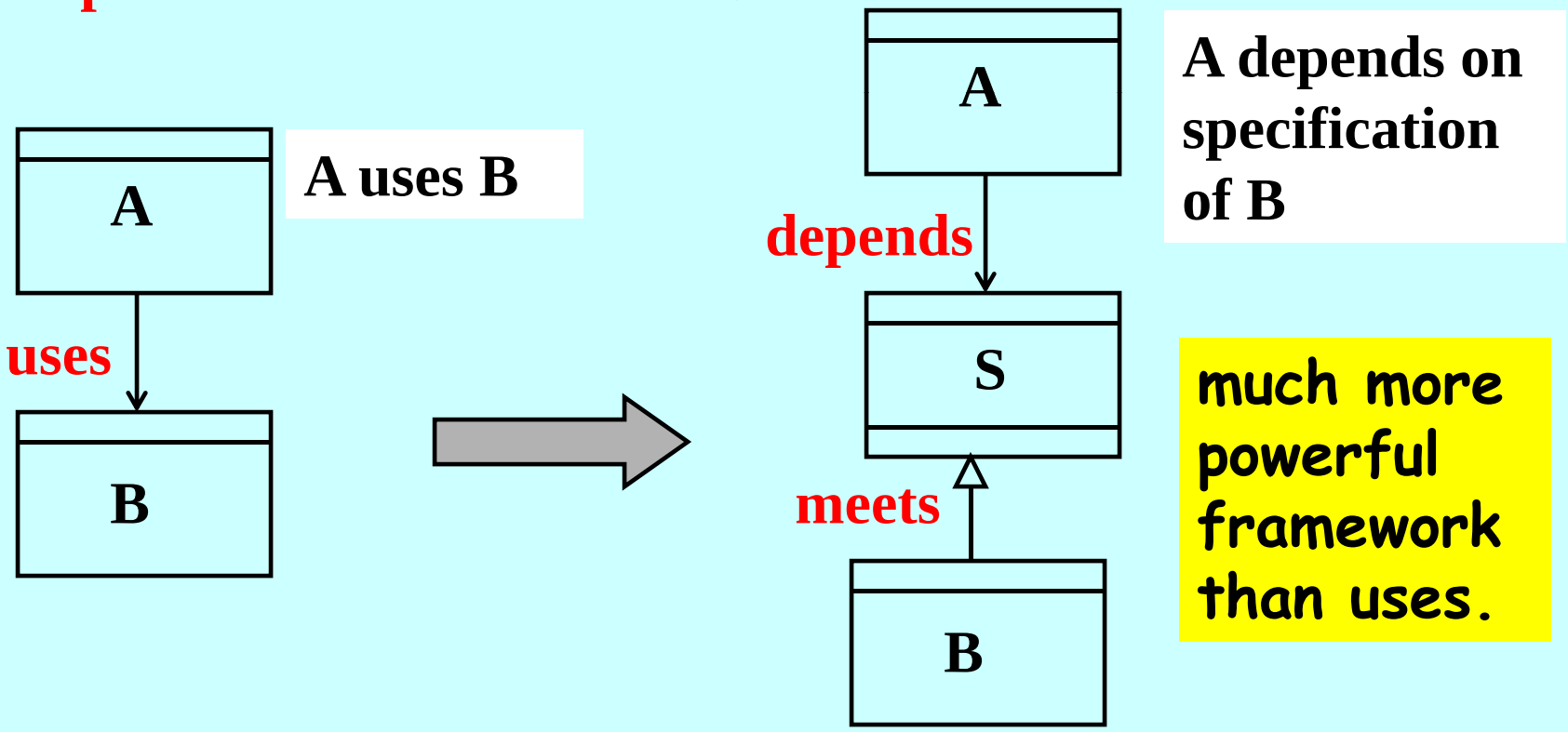
Specifications

• A specification cannot be executed, so we will need for each specification part at least one implementation part that behaves according to the specification.

• The *dependency diagram*, therefore has two kinds of arcs. An implementation part may *depend* on a specification part, and it may *fulfill* or *meet* a specification part.

From Uses to Depends

- **We have broken the *uses* relationship between two parts *A* and *B* into two separate relationships.**
  - **By introducing a specification part *S*, we can say that *A* depends on *S* and *B* meets *S*.**



Advantages of Depends Relationship

The introduction of specifications brings many advantages:

• ***Weakened Assumptions****.* When *A* uses *B,* it is unlikely to rely on every aspect of *B.* Specifications allows us to say which aspects matter and to reason about correctness of parts easily.

• ***Evaluating Changes:*** The specification S helps limit the scope of a change. Suppose we want to change *B.* Must A change as well? Now this question doesn't require looking at A. We start by looking at S, the specification A requires of the part it uses. If the new *B* will still meet S, then no change to A will be needed at all.

Advantages of Depends Relationship (contd.)

• ***Communication***. If A and *B* are to be built by different people, they only need to agree upon S in advance. A can ignore the details of the services *B* provides, and *B* can ignore the details of the needs of A.

• ***Multiple Implementations***. There can be many different implementation modules that meet a given specification part. This makes a market in interchangeable modules possible. Modules are marketed in a catalog by the specifications they meet, and a customer can pick any module that meets the required specification.

• **Good design should have a specification part corresponding to every implementation part.**

•**Java provides some useful mechanisms for expressing decoupling with specifications.**

Interface ??

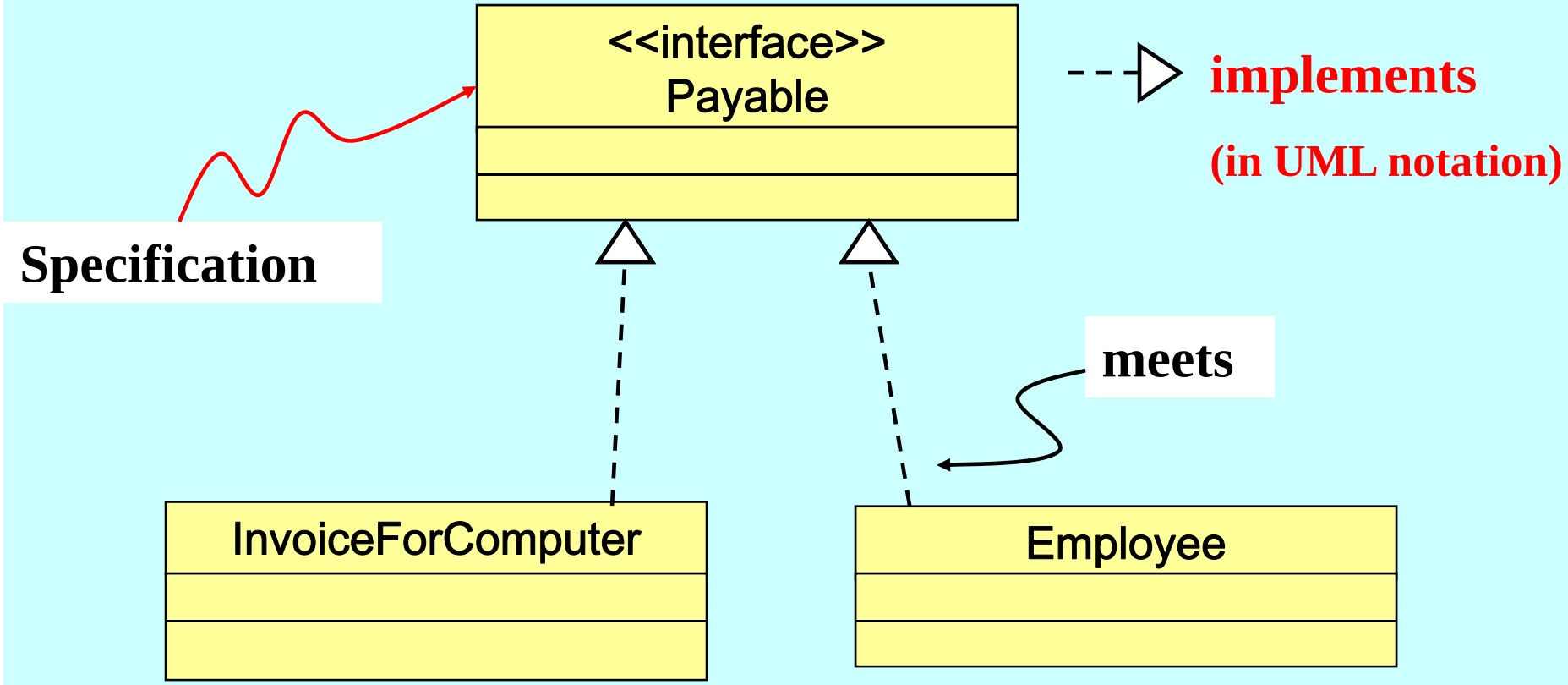
• **Design patterns make extensive use of specifications.**

Interface

- **can contain only constants and abstract methods**

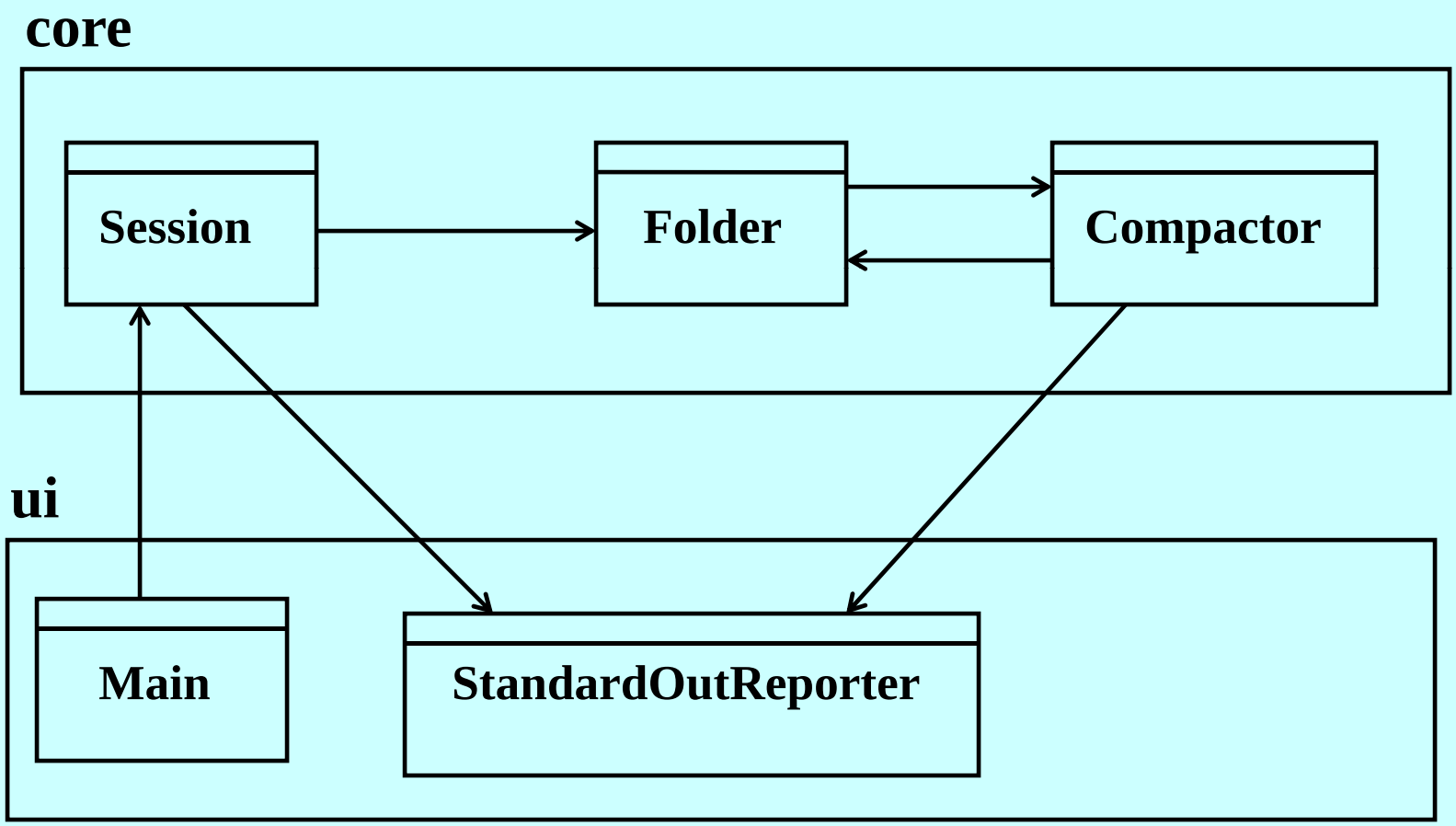
```
public interface InterfaceName
{
    //constants and abstract methods
}
```

Classes can implements interfaces



Decoupling with Interfaces: Example

- **Email client**



Email client example (contd.)

```
public class StandardOutReporter {
    public static void report(String msg) {
        System.out.println(msg);
    }
}
```

In *download* method of Session class,

```
public void download(…) {
    StandardOutReporter.report(“Starting Download”);
}
```

In *main* method of Main class,

```
Session s = new Session(…);
s.download(…);
```

Email client example (contd.)

• If we want to create a new version of our software with a graphical user interface (GUI), we will need to replace the StandardOutReporter class with one that contains appropriate GUI code. This would mean:

- **changing all the references in the core package to refer to a different class, OR**
- **changing the code of the class itself and maintaining two incompatible versions of the class with same name.**

Neither of these is an attractive option.

Email client example (contd.)

```
public interface Reporter {
    void report(String msg);
}

public class StandardOutReporter implements
Reporter {
    public void report(String msg) {
        System.out.println(msg);
    }
}
```

Email client example (contd.)

In *download* method of Session class,

```
public void download(Reporter r, …) {
    r.report(“Starting Download”);
}
```

In *main* method of Main class,

```
Session s = new Session(…);
s.download(new StandardOutReporter(), …);
```

Email client example (contd.)