

HTTP Protocol Implementation

RES, Lecture 4bis

Olivier Liechti

heig-vd

Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

RFC 2616 (HTTP/1.1) is now obsolete

An effort to revise HTTP/1.1 started in 2006, which led to the creation of the IETF httpbis Working Group. Work completed with the publication of RFC 723X (See below)

2014-06 RFC 723X published

RFC7230: HTTP/1.1, part 1: Message Syntax and Routing

RFC7231: HTTP/1.1, part 2: Semantics and Content

RFC7232: HTTP/1.1, part 4: Conditional Requests

RFC7233: HTTP/1.1, part 5: Range Requests

RFC7234: HTTP/1.1, part 6: Caching

RFC7235: HTTP/1.1, part 7: Authentication

Along with:

RFC7236: Initial HTTP Authentication Scheme Registrations

RFC7237: Initial HTTP Method Registrations

And related specifications:

RFC7238: The HTTP Status Code 308 (Permanent Redirect)

RFC7239: Forwarded HTTP Extension

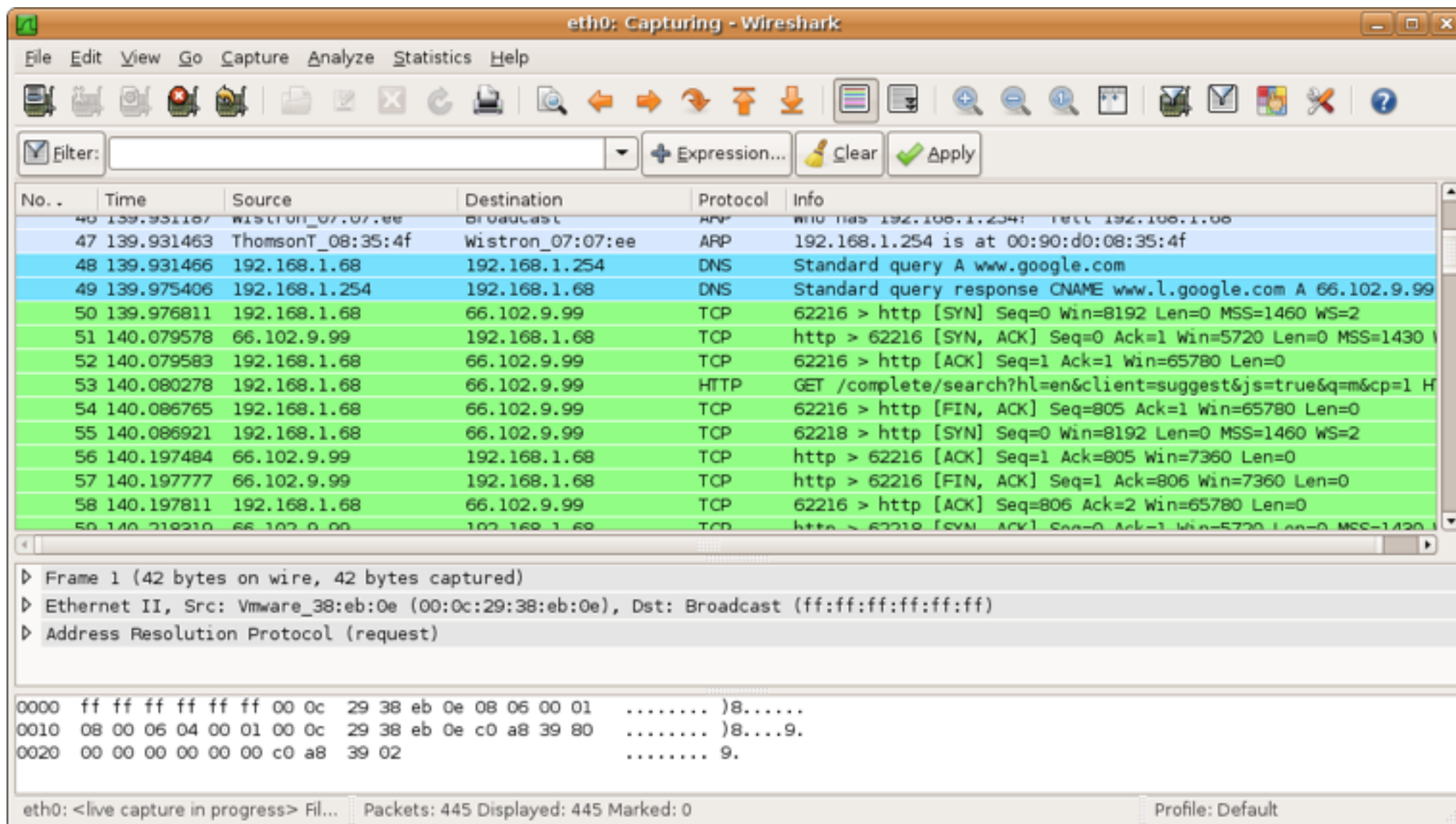
RFC7240: Prefer Header for HTTP

Wireshark



heig-vd

Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud



- Imagine that you are in a team who is implementing a **new web browser** (with a GUI), a **web crawler** (such as search engine robot) or any other kind of **HTTP client**.
- You have designed the high-level architecture of your software and **distributed responsibilities between modules**. Some modules focus on the UI, others on the business logic, others on the storage.
- One module is responsible for dealing with communication protocols, i.e. for implementing the HTTP protocol. You are in charge of this module and you have to **design, implement and test it**. We will refer to this module as the **HTTP Client Library**.
- For the purpose of the exercise, you cannot use any existing HTTP client library. You cannot use the standard `URLConnection` class either. You have to use the `Socket` class and **take care of all details of the HTTP protocol**.

Browser GUI



*“The user has clicked on a link... **GET** the resource located at this URL and give me data that I will render.”*

*“The user has submitted a form. **POST** data to this URL and give me a response.”*

Http Client Library

Development process

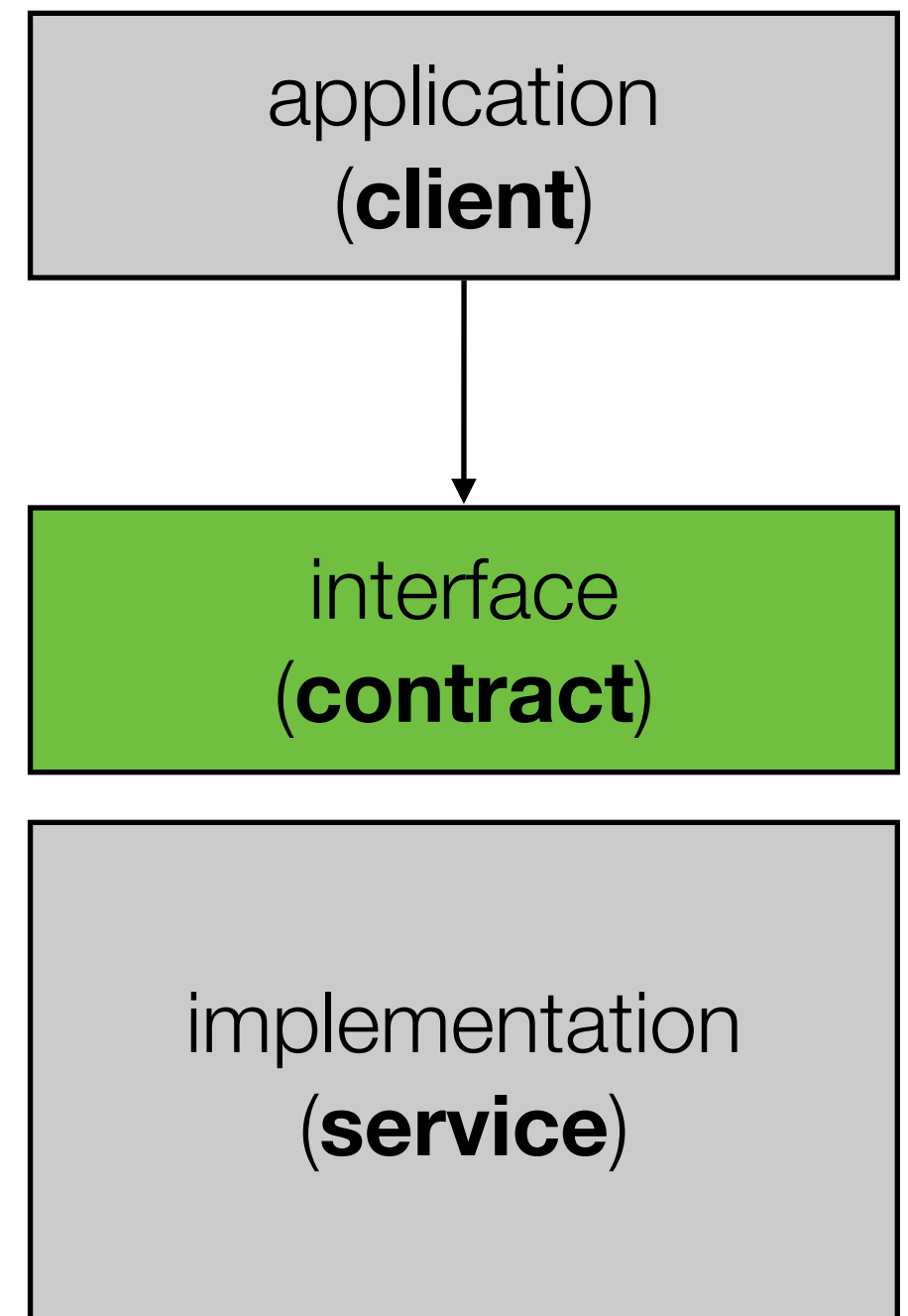
- **Task 1: define the contract** (i.e. the interface) between your HTTP client module and the rest of the application (e.g. the UI). Define a set of interfaces and select their names carefully. For each interface, define the methods and their signatures.
- **Task 2: write a set of JUnit tests**, not only to make it possible to validate the behavior of your code (also as you refactor it), but also to guide you during the design of your interfaces.
- **Task 3: develop classes that implement your interfaces** (or that are private helper classes).
- The tasks 1, 2 and 3 are **not strictly sequential**: it is difficult to get the best interface design immediately, so you will go through an **iterative process**.

Planning

- Step 1: **define the contract** (core interfaces) (10') + review (5')
- Step 2: **refine the contract** (syntax & data structure) (15') + review (5')
How to deal with HTTP headers
- Step 3: **implement some JUnit tests** (10') + review (10')
HTTP status codes
- Step 4: **implement HttpRequest** (15') + review (10')
The Host header & connections
- Step 5: **implement HttpResponse v0.1** (15')
- Step 6: **refactor HttpResponse** into v0.2 (15')
- Step 7: **refactor HttpResponse** (60') + review
- **Final step: finish the implementation and validate with JUnit tests (x')**

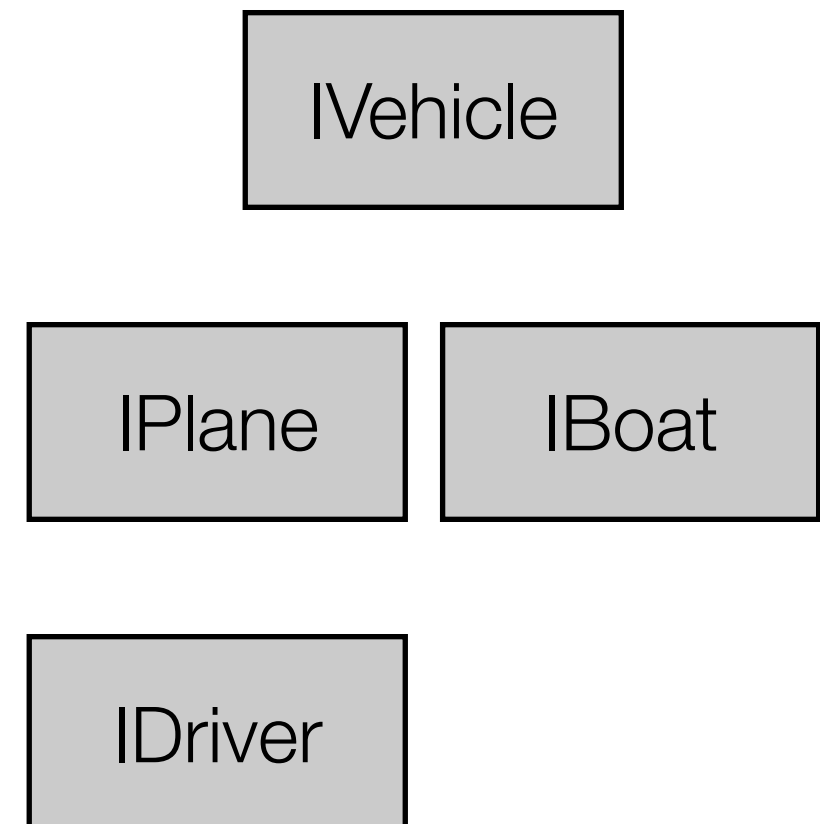
Define the contract

- To go through the process, imagine that you are developing the **GUI** of a web browser.
- In other words, put you **in the shoes of the client** who will use your module (i.e. the service that you provide).
- When the user enters a URL in the navigation bar and presses enter, what is **the code that you would like to write**?
- The point is when you are building the GUI, you don't want to worry about the implementation of the HTTP protocol. You want a **high-level interface** that allows you to send a request and receive a response.



Step 1: Define the contract (10')

- **Read Section 2.1** of the RFC 7230 (Client/Server Messaging).
- <http://tools.ietf.org/html/rfc7230#section-2.1>
- From this specification, sketch out the core abstractions (i.e. the main interfaces) that you envision for your HTTP client library.
- Don't worry about all the methods at this stage: you should **produce a class diagram with ~4-5 interfaces**.
- You should also define the main method that you expect to call from your GUI code, in order to initiate the request-response process.



Review & discussion (5')

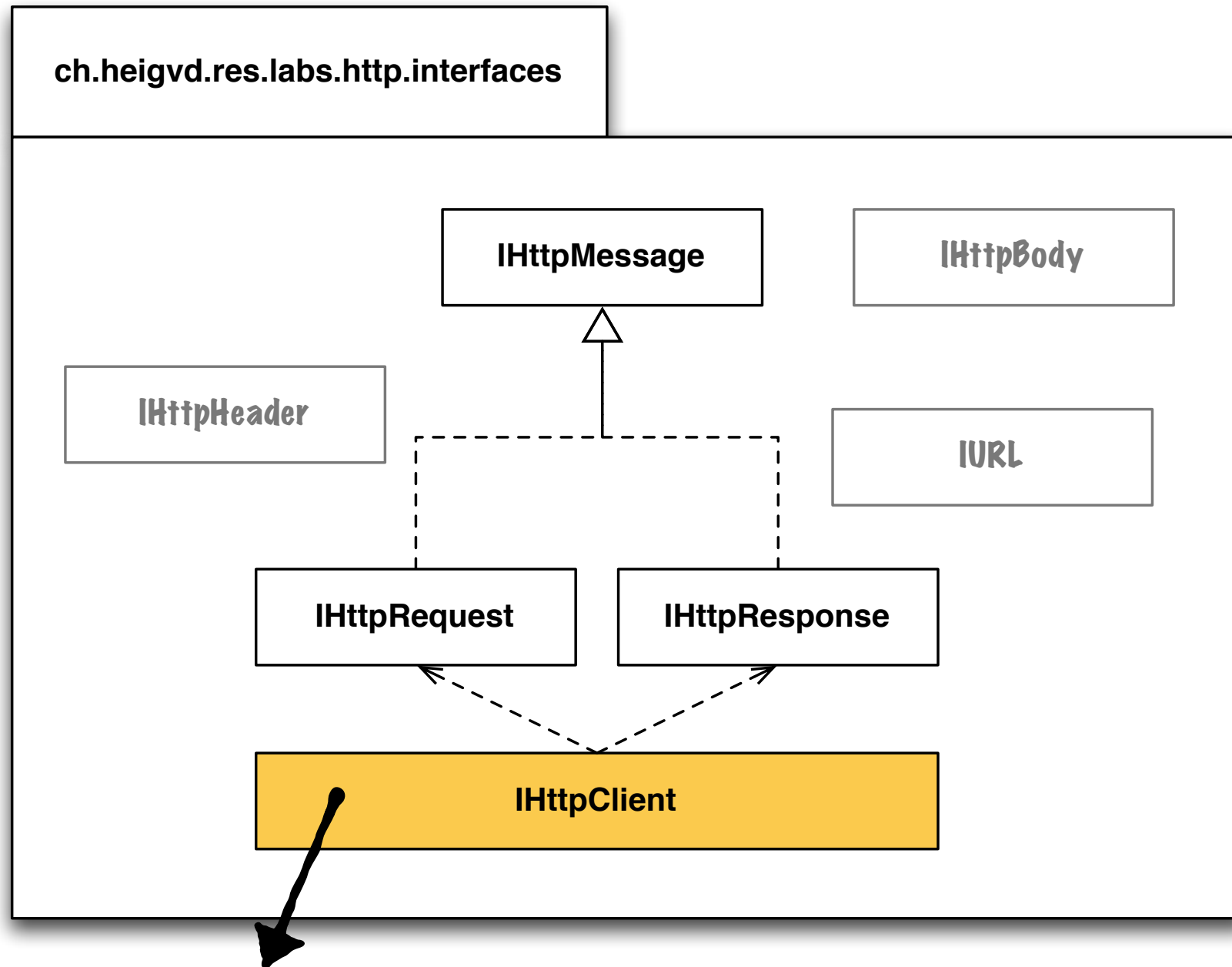
HTTP is a stateless **request/response** protocol that operates by exchanging **messages** (Section 3) across a reliable transport- or session-layer "connection" (Section 6).



A client sends an HTTP request to a server in the form of a **request message**, beginning with a request-line that includes a method, **URI**, and protocol version (Section 3.1.1), followed by **header fields** containing request modifiers, client information, and representation metadata (Section 3.2), an empty line to indicate the end of the header section, and finally a **message body** containing the payload body (if any, Section 3.3).

A server responds to a client's request by sending one or more HTTP response messages, each beginning with a status line that includes the protocol version, a success or error code, and textual reason phrase (Section 3.1.2), possibly followed by header fields containing server information, resource metadata, and representation metadata (Section 3.2), an empty line to indicate the end of the header section, and finally a message body containing the payload body (if any, Section 3.3).

Review & discussion

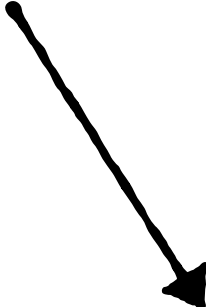


```
public IHttpResponse sendRequest(IHttpRequest request);
```

Step 2: Refine the contract (15')

getter and setter methods for properties

- **Read the following sections:**
 - **3. Message Format**
 - **3.1. Start Line**
 - **3.1.1. Request Line**
 - **3.1.2. Status Line**
 - **3.2. Header fields**
- These sections specify the syntax of the HTTP messages (request and response). Based on this syntax, **define getter methods** in the IHttpMessage, IHttpRequest and IHttpResponse interfaces.



Student
firstName: String lastName: String
getFirstName(): String setFirstName(String) getLastName(): String setLastName(String)

Review & discussion (5')

Each header field consists of a **case-insensitive field name** followed by a **colon (":")**, **optional leading whitespace**, the field value, and optional trailing whitespace.

Content-Length: 12
Content-length: 12
CoNTeNt-LeNGtH:12

When you define the data structure to store headers, make sure that your lookup methods are case insensitive.

Historically, **HTTP header field values could be extended over multiple lines** by preceding each extra line with at least one space or horizontal tab (obs-fold). **This specification deprecates** such line folding except within the message/http media type (Section 8.3.1). A sender **MUST NOT** generate a message that includes line folding

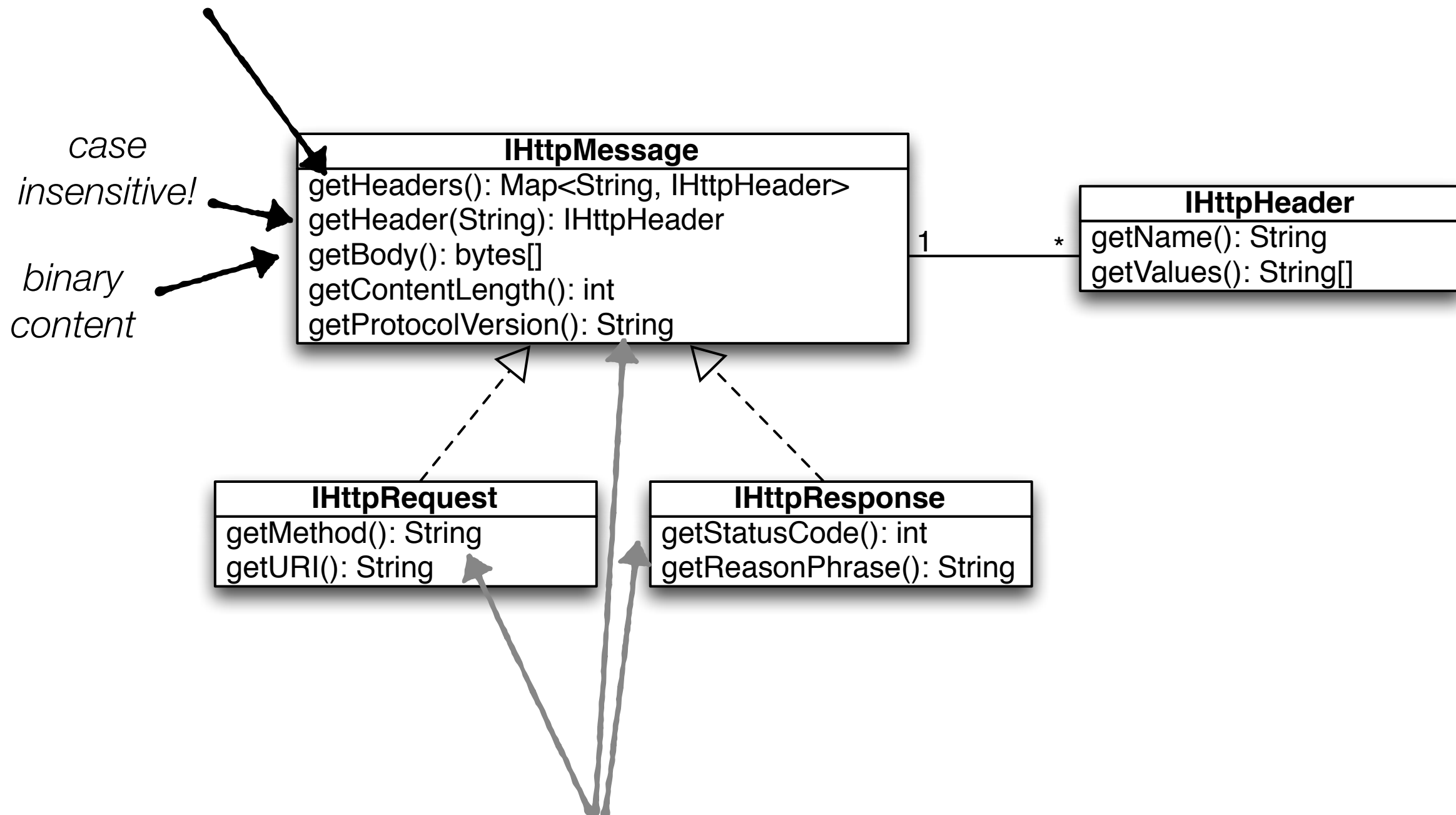
Review & discussion

A sender **MUST NOT** generate multiple header fields with the same field name in a message **unless** either the entire field value for that header field is defined as a comma-separated list [i.e., #(values)] or the header field is a **well-known exception** (as noted below).

Note: In practice, the **"Set-Cookie" header field** ([RFC6265]) often appears multiple times in a response message and does not use the list syntax, **violating the above requirements** on multiple header fields with the same name. Since it cannot be combined into a single field-value, recipients ought to handle "Set-Cookie" as a special case while processing header fields. (See Appendix A.2.3 of [Kri2001] for details.)

Review & discussion

use a Map to easily lookup headers by header names



it would be better to define enum types...

Step 3: Write some JUnit tests (10')

- The **first test** that comes to mind is that it should be possible to use your HTTP client library to send a valid HTTP request (to a known HTTP server) and to receive an HTTP response as a result. **This response should have a status code of 200.**
- Add this unit test in the test project and **resolve the issues** as they appear:
 - So far, we have only defined interfaces. To write the test (so that it compiles), we need an implementation of `IHttpClient`.
 - When we code the implementation of `IHttpClient`, we will also need an implementation of `IHttpRequest` and `IHttpResponse`.
- Add 2 other tests: one where you expect the status code to be **404** and another one where you expect, and one where you expect the status code to be **302**.

Review & discussion (10')

```
@Test
```

```
public void itShouldBePossibleToSendAGETRequest() throws IOException {  
    IHttpClient client = new HttpClient();  
    HttpRequest request = new HttpRequest();  
    request.setMethod("GET");  
    request.setProtocolVersion("HTTP/1.1");  
    request.setURI("http://www.heig-vd.ch/presentation");  
    IHttpResponse response = client.sendRequest(request);  
    assertEquals(200, response.getStatusCode());  
}
```

```
@Test
```

```
public void itShouldBePossibleToHandleA302StatusCode() throws IOException {  
    IHttpClient client = new HttpClient();  
    HttpRequest request = new HttpRequest();  
    request.setMethod("GET");  
    request.setProtocolVersion("HTTP/1.1");  
    request.setURI("http://www.google.com/");  
    IHttpResponse response = client.sendRequest(request);  
    assertEquals(302, response.getStatusCode());  
}
```

Review & discussion

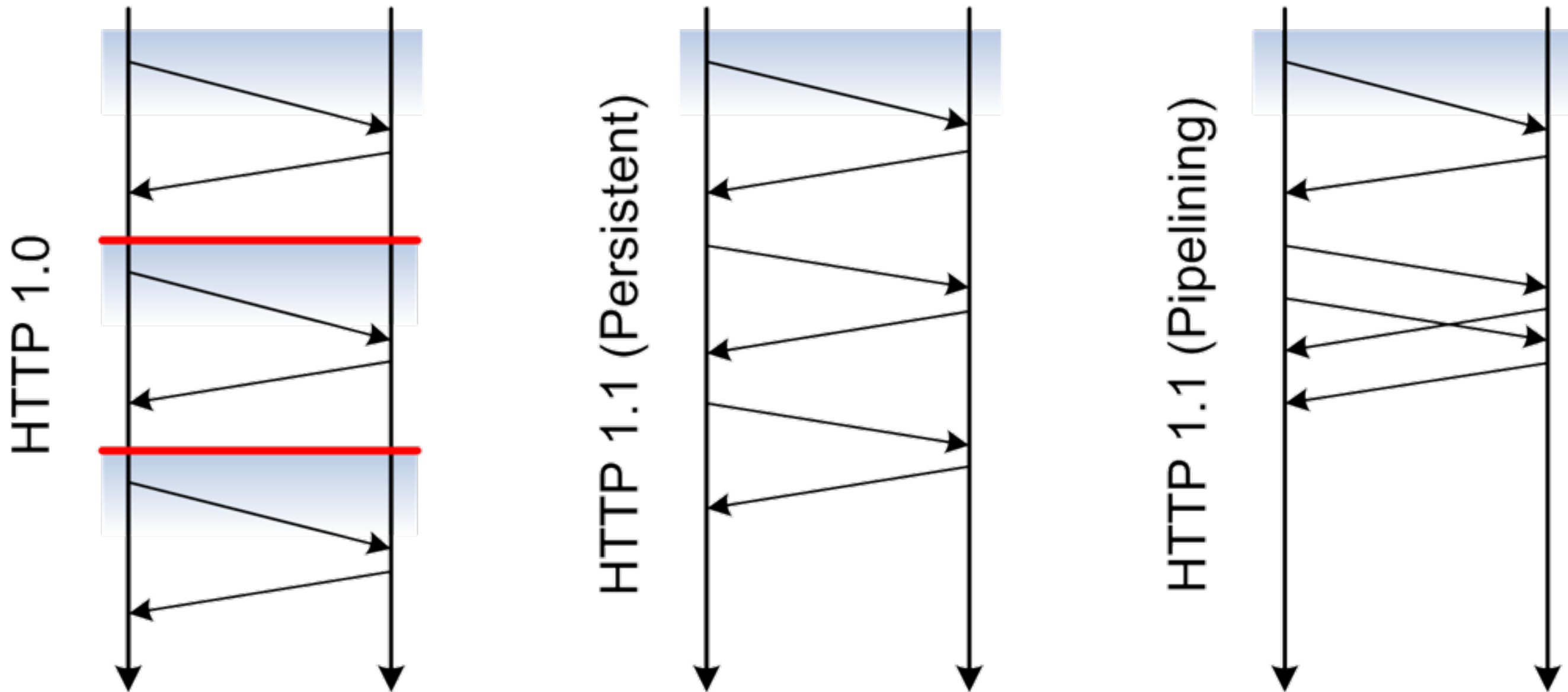
The first digit of the status-code defines the class of response. The last two digits do not have any categorization role. There are five values for the first digit:

- o 1xx (Informational): The request was received, continuing process
- o 2xx (Successful): The request was successfully received, understood, and accepted
- o 3xx (Redirection): Further action needs to be taken in order to complete the request
- o 4xx (Client Error): The request contains bad syntax or cannot be fulfilled
- o 5xx (Server Error): The server failed to fulfill an apparently valid request

Step 4: implement `HttpRequest` (15')

- Implement **constructor(s)** and **setter** methods.
- Use the **standard URI class** to parse the target URL (and extract the hostname, the port, the URI).
- Open the **socket**, providing the hostname and port extracted from.
- Deal with the **Host header** (in the constructor).
- Write the code to send the HTTP request through the socket (following the HTTP/1.1 syntax).
- **Read the status line** of the response.

HTTP & TCP Connections



[http://dret.net/lectures/web-fall07/foundations#\(20\)](http://dret.net/lectures/web-fall07/foundations#(20))

<http://www.apacheweek.com/features/http11>

What is the Host header?

With HTTP/1.0, the Host header was not mandatory.
What was the problem with this?

```
telnet www.nestle.ch 80
```

```
GET /index.html HTTP/1.0\r\n\r\n
```

What is the Host header?

With HTTP/1.0, the Host header was not mandatory.
What was the problem with this?

```
telnet www.nestle.ch 80
```

```
GET /index.html HTTP/1.0\r\n\r\n
```

1. DNS resolution: `www.nestle.ch` -> `192.x.x.x`
2. TCP connection request (on the IP address)
3. Relative URI
4. Is it possible to host more than one brand web site on the same IP?

What is the Host header?

The Host header is used for **multiplexing**. It makes it possible to create **virtual hosts** on a single IP address.

```
telnet www.nestle.ch 80
```

```
GET /index.html HTTP/1.0\r\n  
Host: www.nestle.ch  
\r\n
```

```
telnet www.cardinal.ch 80
```

```
GET /index.html HTTP/1.0\r\n  
Host: www.cardinal.ch  
\r\n
```

“if I look at the Host header, I know
which resource to fetch!”

192.168.42.42

nestle

nike

cardinal

Exercise 5: HttpResponse (v0.1) (15')

- Add a **no-arg constructor**(s) and **setter** methods.
- Use this class in `HttpClient` to create an instance with the status code and return it to the caller.
- **Run the JUnit tests: the 3 tests should be green** (although you only have a partial implementation).

Exercise 6: refactor HttpResponse (15')

- Add a **constructor**, with one argument of type `InputStream`.
- This constructor is supposed to work like this: you pass it an input stream and **the constructor starts parsing** data from the stream. At the end, the different properties (status code, reason phrase, headers, body, etc.) are available and can be retrieved by **getters**.
- **Start with minimal (and incomplete logic)**: just read the status line sent by the server (that is all you need to get the status code and make the tests pass).
- **This is an example of refactoring**: we have changed the structure of the code without altering its behavior (the unit tests are what we use to assess that the behavior has not changed: the more we have, the more confident we can be).

Exercise 7: refactor `HttpResponse` (60')

- Introduce a `HttpInputStream` **class**, providing a `readHttpLine` method (that uses the `\r\n` delimiters when parsing the messages).
- Add **JUnit tests to make assertions on the HTTP headers and the body** that you expect in responses from well known servers (e.g. check the length of the body).
- Implement the code to **parse the headers and store** them in the map data structure.
- Implement the code to **parse the body**. You have to deal with 3 main cases:
1) you have received a Content-Length header, 2) you have received a Transfer-Encoding: chunked header and 3) you have not received these headers, but the server closes the connection when it has sent the body.