Design Patterns and Refactoring

Course of Software Engineering II
A.A. 2011/2012

Valerio Maggio, PhD Student
Prof. Marco Faella

Starting Scenario

- "We have a DeviceManager that have to handle objects that is able to connect to the GPS Network"
- Objectives of current lecture:
  - Improve and complicate the starting scenario
    - Through refactoring and patterns
  - Interactive Improvements
    - Let's do it together
- As usual let's do Program Comprehension first

Step 1: Extend Controllers

Step 1: Extend Controllers

- **Q:** We want to add a new *type* of Controller
  - `class InternalGalileoController`

Step 1: Extend Controllers

- **Q:** We want to add a new *type* of Controller
  - `class InternalGalileoController`
- Let's look at UML:
  - What do you think about extensibility?
    - Please focus on `InternalGpsConnector`

Step 1: Extend Controllers

- **Q:** We want to add a new *type* of Controller
  - `class InternalGalileoController`
- Let's look at UML:
  - What do you think about extensibility?
    - Please focus on `InternalGpsConnector`

Step 1: Extend Controllers

- **Q:** We want to add a new *type* of Controller
  - `class InternalGalileoController`
- Let's look at UML:
  - What do you think about extensibility?
    - Please focus on `InternalGpsConnector`
- **A: (Refactoring)**
  - **Extract Interface**

Step 2: Client Association

Step 2: Client Association

- **Q:** Direct association between Client and Product
  - Too much **coupling**

Step 2: Client Association

- **Q:** Direct association between Client and Product
  - Too much **coupling**
- Let's look at the code:
  - Where do you think is the “coupling point”?

Step 2: Client Association

- **Q:** Direct association between Client and Product
  - Too much **coupling**
- Let's look at the code:
  - Where do you think is the “coupling point”?
- **A [1]:**
  - Collection of Super-Type Istances

Step 2: Client Association

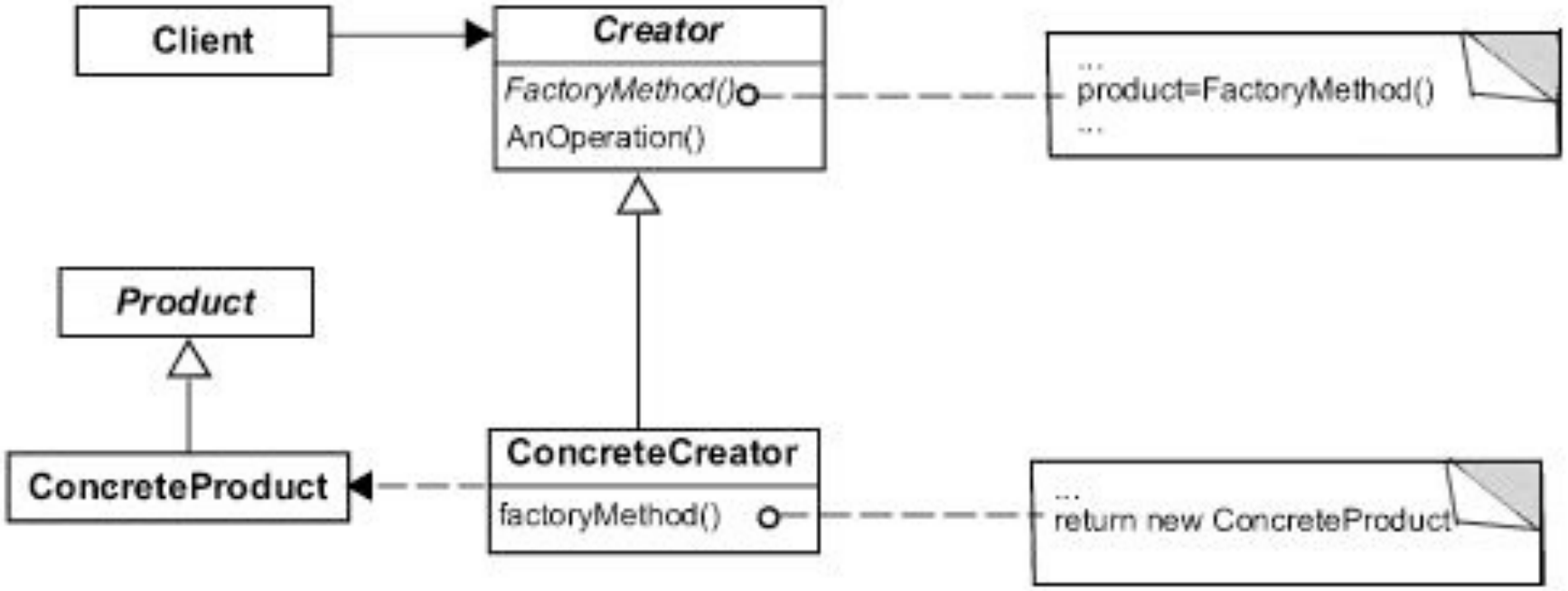
- **Q:** Direct association between Client and Product
  - Too much **coupling**
- Let's look at the code:
  - Where do you think is the “coupling point”?
- **A [1]:**
  - Collection of Super-Type Istances
- **A [2] (design pattern) :**
  - **Factory Method**

Factory Method Pattern

- Intent:
  - Define an interface for creating an object
    - Factory Method let's a class defer instantiation to subclasses.
  - Defining a “virtual” constructor.
  - The new operator considered harmful.

- Needs to standardize the architectural model for a range of products,
  - Allow for individual applications to define their own domain objects and provide for their instantiation.

Factory Method Pattern



Step 3: Improvements

- Let's think about current design
  - Brainstorming please :)
- **Q:** Instantiation of Factory
  - **A (Pattern): Singleton**

Step 3: Improvements

- Let's think about current design
  - Brainstorming please :)
- **Q:** Instantiation of Factory
  - **A (Pattern): Singleton**
- **Q:** “Virtual Constructor Methods”
  - **A:** Multiple methods vs Single Method

Step 4: Requirement Extension

Step 4: Requirement Extension

- We want to add a new *family of connectors*
  - Current products: *Smartphone Connectors*
    - *Gps and Galileo connections*
  - **New Products:** *Mobile Connectors*
    - *Bluetooth and IRDA connections*

Step 4: Requirement Extension

- We want to add a new *family of connectors*
  - Current products: *Smartphone Connectors*
    - *Gps and Galileo connections*
  - **New Products:** *Mobile Connectors*
    - *Bluetooth and IRDA connections*
- **Q:** How to handle creation loosely coupled with client?

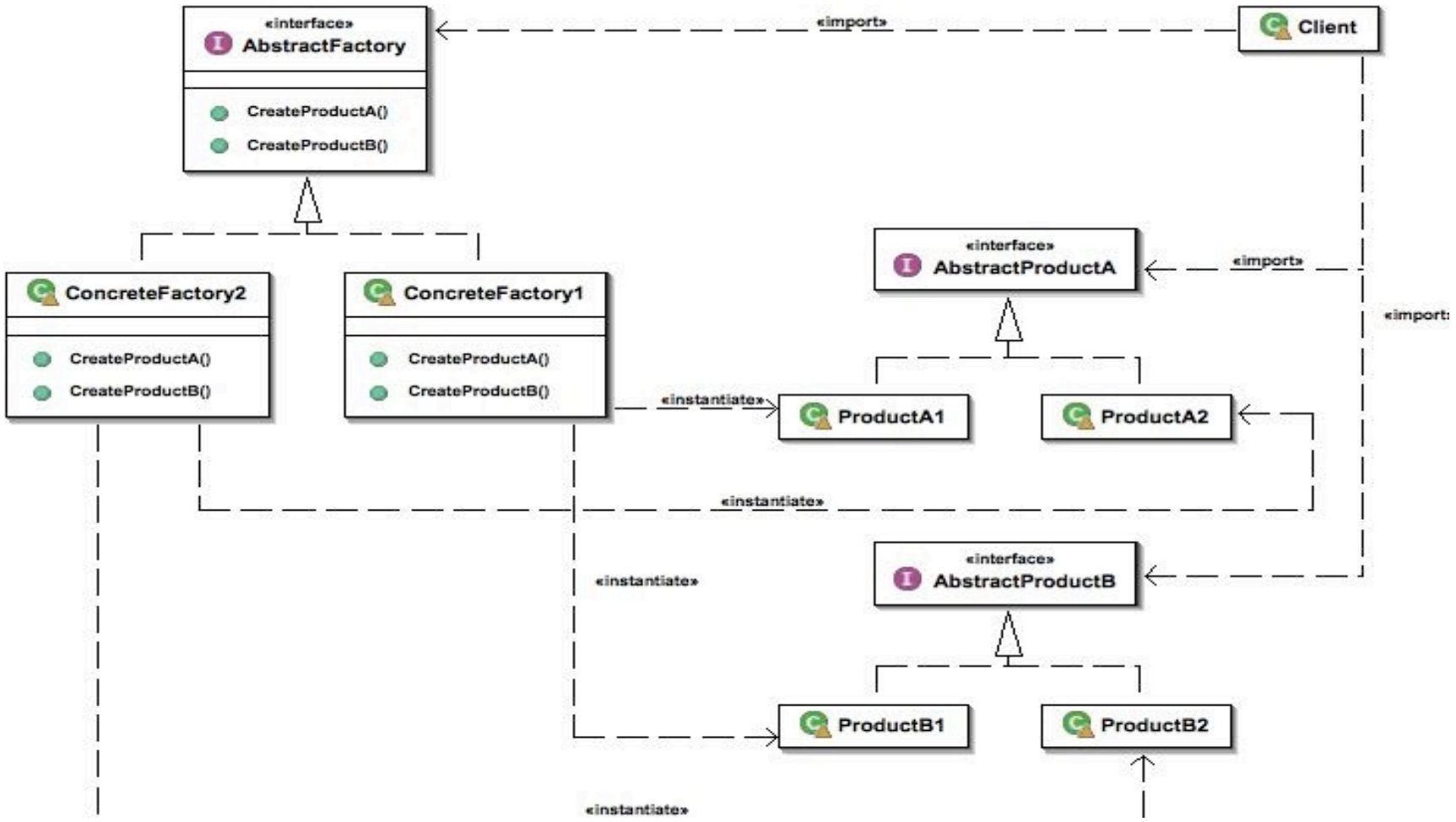
Step 4: Requirement Extension

- We want to add a new *family of connectors*
  - Current products: *Smartphone Connectors*
    - *Gps and Galileo connections*
  - **New Products:** *Mobile Connectors*
    - *Bluetooth and IRDA connections*
- **Q:** How to handle creation loosely coupled with client?
- **A (Pattern):**
  - **Abstract Factory**

Abstract Factory Pattern

- **Intent:**
  - Provide an interface for creating families of related or dependent objects
    - Without specifying their concrete classes.
  - A hierarchy that encapsulates
    - Many possible “platforms”
    - Construction of a suite of “products”.
- **Problem:**
  - An application has to be portable
    - Encapsulate platform dependencies.

Abstract Factory Pattern



Step 4.1: Example

- **Q:** Change Product Family associated to Device Controller
- What is the effort?

Step 4.1: Example

- **Q:** Change Product Family associated to Device Controller
- What is the effort?
- *Minimum effort, maximum effect*
  - *Client loosely coupled with products*
    - *Instantiation and handling*

Step 5: Improvement

Step 5: Improvement

- Improvement in Product instantiation

Step 5: Improvement

- Improvement in Product instantiation
- Extension to new product family:
  - Client point of view: **Easy**
  - Product point of view: **???**

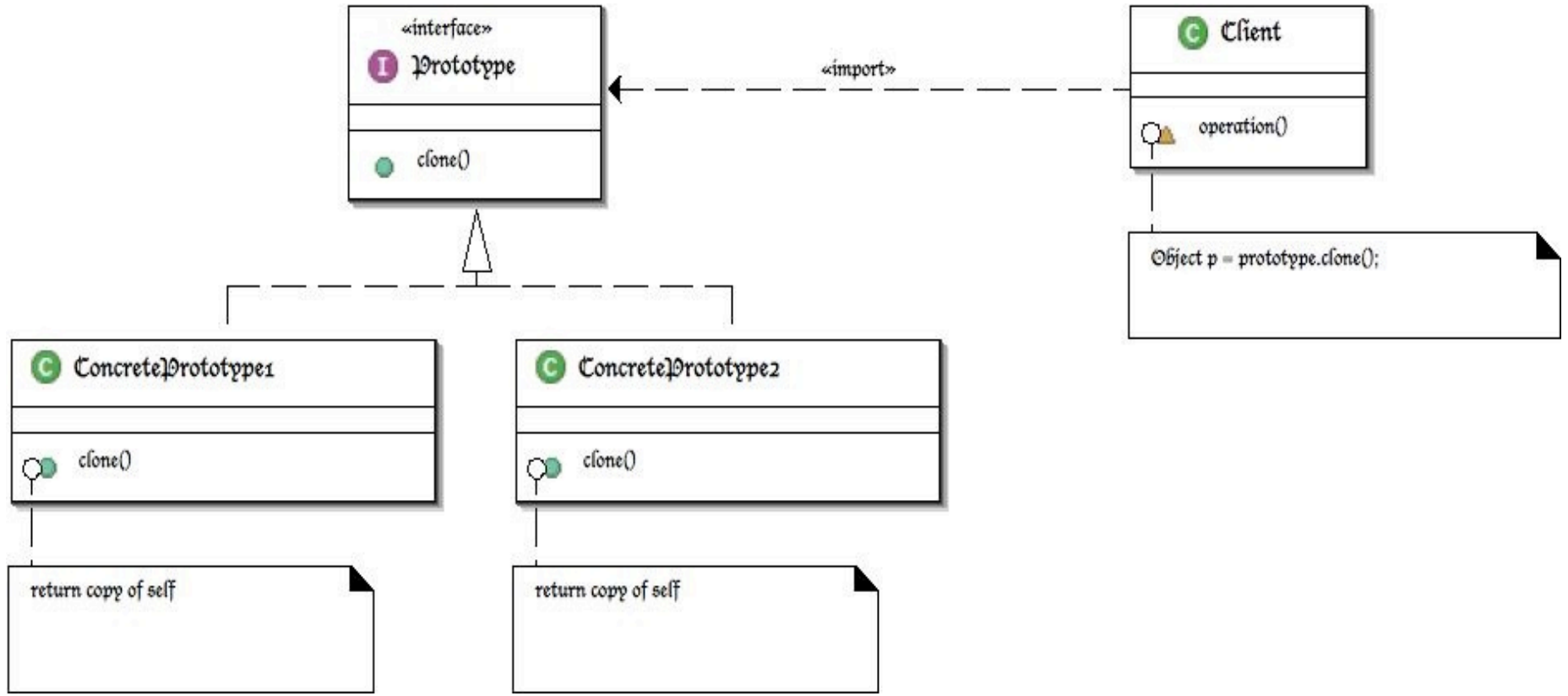
Step 5: Improvement

- Improvement in Product instantiation
- Extension to new product family:
  - Client point of view: **Easy**
  - Product point of view: **???**
- **A (Pattern):**
  - **Prototype**

Prototype Pattern

- **Intent:**
  - Specify the kinds of objects to create using a prototypical instance
    - create new objects by copying this prototype.
  - Co-opt one instance of a class for use as a breeder of all future instances.
- **Problem:**
  - Application “hard wires” the class of object to create in each “new” expression.

Prototype Pattern



Rules of Thumbs

- Sometimes creational patterns are competitors
- Often, designs:
  - Start out using Factory Method
    - (less complicated, more customizable, subclasses proliferate)
  - Evolve toward
    - Abstract Factory
    - Prototype
    - Builder (more flexible, more complex)
- **Don't abuse on using Design Patterns!!**

References

- Gamma, E., Helm, R., Johnson, R. e Vlissides, J., **Design Patterns: Elements of Reusable Object-Oriented Software**

- http://www.artima.com/lejava/articles/designprinciples.html