Creational Design Patterns

Software Modeling Foundations

Author: Eng. Carlos Andrés Sierra, M.Sc.
cavirguezs@udistrital.edu.co

Lecturer
Computer Engineer
School of Engineering
Universidad Distrital Francisco José de Caldas

2024-III

Outline

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Outline

Basic Concepts

- **Intent**: Separate the construction of a complex object from its representation so that the **same construction process** can create **different representations**.
- **Motivation**:
  - **Problem**: An application needs to **create instances** of a class, but the class is abstract and has many **possible implementations**.
  - **Solution**: Provide different object creation mechanisms, which **allow** the client to **create the object** without knowing the actual implementation. This increase flexibility and reuse of code.

Basic Concepts

- **Intent**: Separate the construction of a complex object from its representation so that the **same construction process** can create **different representations**.
- **Motivation**:
  - **Problem**: An application needs to **create instances** of a class, but the class is abstract and has many **possible implementations**.
  - **Solution**: Provide different object creation mechanisms, which **allow** the client to **create the object** without knowing the actual implementation. This increase flexibility and reuse of code.

Basic Concepts

- **Intent**: Separate the construction of a complex object from its representation so that the **same construction process** can create **different representations**.
- **Motivation**:
  - **Problem**: An application needs to **create instances** of a class, but the class is abstract and has many **possible implementations**.
  - **Solution**: Provide different object creation mechanisms, which **allow** the client to **create the object** without knowing the actual implementation. This increase flexibility and reuse of code.

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Outline

Outline

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Builder Pattern — Concepts

- It is a pattern that lets **construct** a complex object **step by step**. The idea is to create **different representations** of an object using the same construction code.
- One typical **problem** is work with a class that has **many attributes** and it is difficult to create an instance of it. It gets worse when there are many possible representations of the object.
- Several **attributes** are **optional** and the client has to specify them in a specific order. So, this could be a problem for both objects management and code maintenance. Also, increase memory consumption.
- The **solution** is to encapsulate the object construction and use separate **methods** to add or build the object attributes.

Builder Pattern — Concepts

- It is a pattern that lets **construct** a complex object **step by step**. The idea is to create **different representations** of an object using the same construction code.
- One typical **problem** is work with a class that has **many attributes** and it is difficult to create an instance of it. It gets worse when there are many possible representations of the object.
- Several **attributes** are **optional** and the client has to specify them in a specific order. So, this could be a problem for both objects management and code maintenance. Also, increase memory consumption.
- The **solution** is to encapsulate the object construction and use separate **methods** to add or build the object attributes.

Builder Pattern — Concepts

- It is a pattern that lets **construct** a complex object **step by step**. The idea is to create **different representations** of an object using the same construction code.
- One typical **problem** is work with a class that has **many attributes** and it is difficult to create an instance of it. It gets worse when there are many possible representations of the object.
- Several **attributes** are **optional** and the client has to specify them in a specific order. So, this could be a problem for both objects management and code maintenance. Also, increase memory consumption.
- The **solution** is to encapsulate the object construction and use separate **methods** to add or build the object attributes.

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Builder Pattern — Concepts

- It is a pattern that lets **construct** a complex object **step by step**. The idea is to create **different representations** of an object using the same construction code.
- One typical **problem** is work with a class that has **many attributes** and it is difficult to create an instance of it. It gets worse when there are many possible representations of the object.
- Several **attributes** are **optional** and the client has to specify them in a specific order. So, this could be a problem for both objects management and code maintenance. Also, increase memory consumption.
- The **solution** is to encapsulate the object construction and use separate **methods** to add or build the object attributes.

Builder Pattern — Classes Structure

Lets the director orchestrate the building process.

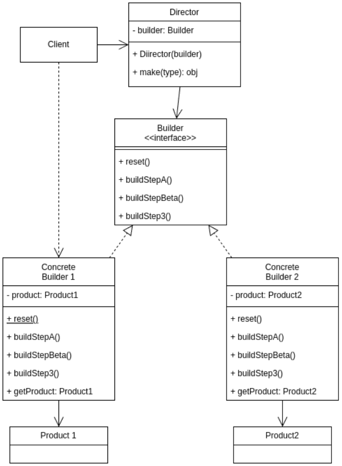

Figure: Builder Pattern Class Diagram

Builder Pattern Example: `Computers`

Outline

Factory Pattern — Concepts

- It is pattern based on a **superclass** and the subclasses could alter the type of objects to be created.
- One of the **most common** design pattern used, is simple, powerful and flexible. It is used with many other **design patterns**.
- It lets make **simple** a complex code. If you have **groups** of **objects** that are created in a similar way, the **factory method** is the best choice.
- The **client** just needs to interact with the **factory**, and the **factory** will create the object. The **client** does **not** need to **know** the actual implementation of the object (or the *subclasses*).

Factory Pattern — Concepts

- It is pattern based on a **superclass** and the subclasses could alter the type of objects to be created.
- One of the **most common** design pattern used, is simple, powerful and flexible. It is used with many other **design patterns**.
- It lets make **simple** a complex code. If you have **groups** of **objects** that are created in a similar way, the **factory method** is the best choice.
- The **client** just needs to interact with the **factory**, and the **factory** will create the object. The **client** does **not** need to **know** the actual implementation of the object (or the *subclasses*).

Factory Pattern — Concepts

- It is pattern based on a **superclass** and the subclasses could alter the type of objects to be created.
- One of the **most common** design pattern used, is simple, powerful and flexible. It is used with many other **design patterns**.
- It lets make **simple** a complex code. If you have **groups** of **objects** that are created in a similar way, the **factory method** is the best choice.
- The **client** just needs to interact with the **factory**, and the **factory** will create the object. The **client** does **not** need to **know** the actual implementation of the object (or the *subclasses*).

Factory Pattern — Concepts

- It is pattern based on a **superclass** and the subclasses could alter the type of objects to be created.
- One of the **most common** design pattern used, is simple, powerful and flexible. It is used with many other **design patterns**.
- It lets make **simple** a complex code. If you have **groups** of **objects** that are created in a similar way, the **factory method** is the best choice.
- The **client** just needs to interact with the **factory**, and the **factory** will create the object. The **client** does **not** need to **know** the actual implementation of the object (or the *subclasses*).

Factory Pattern — Classes Structure

It is like to watch Charlie and the Chocolate Factory.

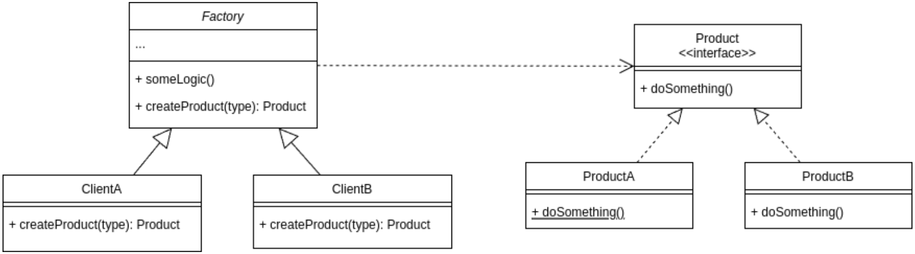

Figure: Factory Pattern Class Diagram

Factory Pattern Example: `On-line Store`

Outline

Abstract Factory Pattern — Concepts

- This is a pattern that lets you **produce families** of related objects without specifying their concrete classes.
- It is a **super factory** that creates other factories. It is used when you have a **super class** that can create **subclasses** and the **subclasses** can create **objects**.
- Also this pattern allows to keep the **client** code decoupled from the **actual objects** in the system. Keep old code when you need to add new representations.
- It is used when you have **many objects** that can be grouped in **families**.

Abstract Factory Pattern — Concepts

- This is a pattern that lets you **produce families** of related objects without specifying their concrete classes.
- It is a **super factory** that creates other factories. It is used when you have a **super class** that can create **subclasses** and the **subclasses** can create **objects**.
- Also this pattern allows to keep the **client** code decoupled from the **actual objects** in the system. Keep old code when you need to add new representations.
- It is used when you have **many objects** that can be grouped in **families**.

Abstract Factory Pattern — Concepts

- This is a pattern that lets you **produce families** of related objects without specifying their concrete classes.
- It is a **super factory** that creates other factories. It is used when you have a **super class** that can create **subclasses** and the **subclasses** can create **objects**.
- Also this pattern allows to keep the **client** code decoupled from the **actual objects** in the system. Keep old code when you need to add new representations.
- It is used when you have **many objects** that can be grouped in **families**.

Abstract Factory Pattern — Concepts

- This is a pattern that lets you **produce families** of related objects without specifying their concrete classes.
- It is a **super factory** that creates other factories. It is used when you have a **super class** that can create **subclasses** and the **subclasses** can create **objects**.
- Also this pattern allows to keep the **client** code decoupled from the **actual objects** in the system. Keep old code when you need to add new representations.
- It is used when you have **many objects** that can be grouped in **families**.

Abstract Factory Pattern — Classes Structure

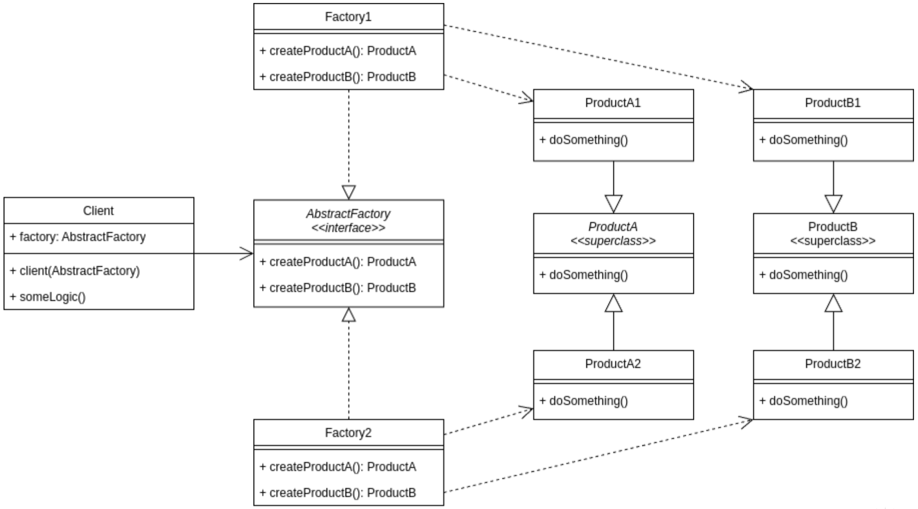

Figure: Abstract Factory Pattern Class Diagram

Abstract Factory Pattern Example: `Furniture Shop`

Outline

Singleton Pattern — Concepts

- In an attemp to **reduce memory consumption**, this pattern ensure that a class has only **one instance** and provide a global point of access to it.
- It is used when you need to **control** the **number of instances** of a class, so just one class instance is allowed across all the application. Also it allows to apply the concept of **lazy creation**.
- It is pretty simple: just create a class with a method that **creates** a

  new **instance** of the class if one does not exist. If an instance already exists, it returns a reference to that object.
- It violates the *Single Responsability Principle* and the *Open/Closed Principle*. Also, internal instance and get method are *static*.
- **Not** a very **good idea** if you are using a multi-trending application, could be issues trying to access a shared single object.

Singleton Pattern — Concepts

- In an attemp to **reduce memory consumption**, this pattern ensure that a class has only **one instance** and provide a global point of access to it.
- It is used when you need to **control** the **number of instances** of a class, so just one class instance is allowed across all the application. Also it allows to apply the concept of **lazy creation**.
- It is pretty simple: just create a class with a method that **creates** a new **instance** of the class if one does not exist. If an instance already exists, it returns a reference to that object.
- It violates the *Single Responsability Principle* and the *Open/Closed Principle*. Also, internal instance and get method are *static*.
- **Not** a very **good idea** if you are using a multi-trending application could be issues trying to access a shared single object.

Singleton Pattern — Concepts

- In an attemp to **reduce memory consumption**, this pattern ensure that a class has only **one instance** and provide a global point of access to it.
- It is used when you need to **control** the **number of instances** of a class, so just one class instance is allowed across all the application. Also it allows to apply the concept of **lazy creation**.
- It is pretty simple: just create a class with a method that **creates** a new **instance** of the class if one does not exist. If an instance already exists, it returns a reference to that object.
- It violates the *Single Responsability Principle* and the *Open/Closed Principle*. Also, internal instance and get method are *static*.
- **Not** a very **good idea** if you are using a multi-trending application could be issues trying to access a shared single object.

Singleton Pattern — Concepts

- In an attemp to **reduce memory consumption**, this pattern ensure that a class has only **one instance** and provide a global point of access to it.
- It is used when you need to **control** the **number of instances** of a class, so just one class instance is allowed across all the application. Also it allows to apply the concept of **lazy creation**.
- It is pretty simple: just create a class with a method that **creates** a

  new **instance** of the class if one does not exist. If an instance already exists, it returns a reference to that object.
- It violates the *Single Responsability Principle* and the *Open/Closed Principle*. Also, internal instance and get method are *static*.
- **Not** a very **good idea** if you are using a multi-trending application could be issues trying to access a shared single object.

Singleton Pattern — Concepts

- In an attemp to **reduce memory consumption**, this pattern ensure that a class has only **one instance** and provide a global point of access to it.
- It is used when you need to **control** the **number of instances** of a class, so just one class instance is allowed across all the application. Also it allows to apply the concept of **lazy creation**.
- It is pretty simple: just create a class with a method that **creates** a

  new **instance** of the class if one does not exist. If an instance already exists, it returns a reference to that object.
- It violates the *Single Responsability Principle* and the *Open/Closed Principle*. Also, internal instance and get method are *static*.
- **Not** a very **good idea** if you are using a multi-trending application, could be issues trying to access a shared single object.

Singleton Pattern — Classes Structure

Think in a circle room with several doors but *just one doorman*.

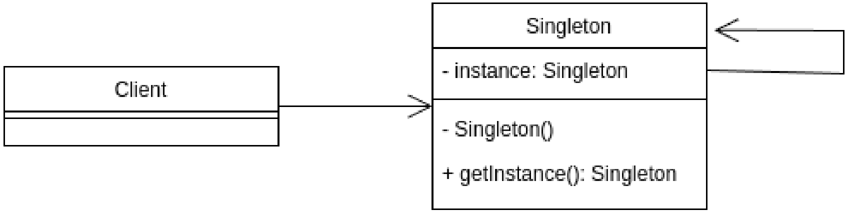

Figure: Singleton Pattern Class Diagram

Singleton Pattern Example: `Game Style Preferences`

Outline

Prototype Pattern — Concepts

- It is based on **copy** of an **existing object**. It is used when the type of objects to create is determined by a prototypical instance, which is **cloned** to produce new objects.
- Remember, **clone** is not just copy an object, it is **create** a new **object** with the **same** attributes and values of the original object.
- It solves the **problem** of copy the private attributes of an object. So, you could create a **copy including the hidden logic**.
- This pattern **delegates** the **cloning** process to the **actual objects** that are being cloned. This is a good idea because the object knows how to create a copy of itself using an internal method.
- It exists the concept of **prototype registry**, just to make quick access and save of frecuent used objects.

Prototype Pattern — Concepts

- It is based on **copy** of an **existing object**. It is used when the type of objects to create is determined by a prototypical instance, which is **cloned** to produce new objects.
- Remember, **clone** is not just copy an object, it is **create** a new **object** with the **same** attributes and values of the original object.
- It solves the **problem** of copy the private attributes of an object. So, you could create a **copy including the hidden logic**.
- This pattern **delegates** the **cloning** process to the **actual objects** that are being cloned. This is a good idea because the object knows how to create a copy of itself using an internal method.
- It exists the concept of **prototype registry**, just to make quick access and save of frecuent used objects.

Prototype Pattern — Concepts

- It is based on **copy** of an **existing object**. It is used when the type of objects to create is determined by a prototypical instance, which is **cloned** to produce new objects.
- Remember, **clone** is not just copy an object, it is **create** a new **object** with the **same** attributes and values of the original object.
- It solves the **problem** of copy the private attributes of an object. So, you could create a **copy including the hidden logic**.
- This pattern **delegates** the **cloning** process to the **actual objects** that are being cloned. This is a good idea because the object knows how to create a copy of itself using an internal method.
- It exists the concept of **prototype registry**, just to make quick access and save of frecuent used objects.

Prototype Pattern — Concepts

- It is based on **copy** of an **existing object**. It is used when the type of objects to create is determined by a prototypical instance, which is **cloned** to produce new objects.
- Remember, **clone** is not just copy an object, it is **create** a new **object** with the **same** attributes and values of the original object.
- It solves the **problem** of copy the private attributes of an object. So, you could create a **copy including the hidden logic**.
- This pattern **delegates** the **cloning** process to the **actual objects** that are being cloned. This is a good idea because the object knows how to create a copy of itself using an internal method.
- It exists the concept of **prototype registry**, just to make quick access and save of frecuent used objects.

Prototype Pattern — Concepts

- It is based on **copy** of an **existing object**. It is used when the type of objects to create is determined by a prototypical instance, which is **cloned** to produce new objects.
- Remember, **clone** is not just copy an object, it is **create** a new **object** with the **same** attributes and values of the original object.
- It solves the **problem** of copy the private attributes of an object. So, you could create a **copy including the hidden logic**.
- This pattern **delegates** the **cloning** process to the **actual objects** that are being cloned. This is a good idea because the object knows how to create a copy of itself using an internal method.
- It exists the concept of **prototype registry**, just to make quick access and save of frecuent used objects.

Prototype Pattern — Classes Structure

You know all my secrets, so you could create a clone of me.

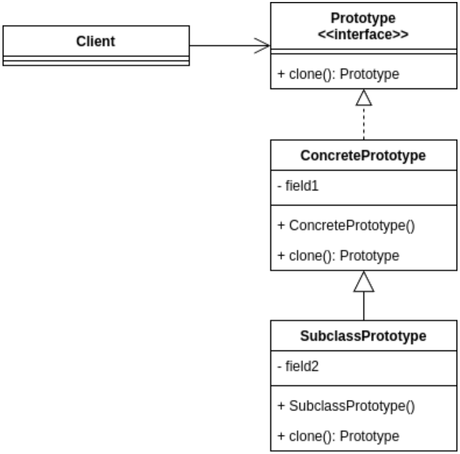

Figure: Prototpe Pattern Class Diagram

Prototype Pattern Example: `Cellular Differenciation`

Outline

Conclusions

- There are a few ways to **create objects** inside an application in a pretty eficient way.You just need to think about it and choose the best one for your application.
- You **could combine** these patterns to create a more complex and flexible application. However, you need to be careful with the complexity of the application.
- The **Builder** pattern is used to create a complex object step by step. The **Factory** pattern is used to create objects in a simple way. The **Abstract Factory** pattern is used to create families of objects. The **Singleton** pattern is used to create just one instance of a class. The **Prototype** pattern is used to create a new object by copying an existing object.

Conclusions

- There are a few ways to **create objects** inside an application in a pretty eficient way.You just need to think about it and choose the best one for your application.
- You **could combine** these patterns to create a more complex and flexible application. However, you need to be careful with the complexity of the application.
- The **Builder** pattern is used to create a complex object step by step. The **Factory** pattern is used to create objects in a simple way. The **Abstract Factory** pattern is used to create families of objects. The **Singleton** pattern is used to create just one instance of a class. The **Prototype** pattern is used to create a new object by copying an existing object.

Conclusions

- There are a few ways to **create objects** inside an application in a pretty eficient way.You just need to think about it and choose the best one for your application.
- You **could combine** these patterns to create a more complex and flexible application. However, you need to be careful with the complexity of the application.
- The **Builder** pattern is used to create a complex object step by step.
  The **Factory** pattern is used to create objects in a simple way.
  The **Abstract Factory** pattern is used to create families of objects.
  The **Singleton** pattern is used to create just one instance of a class.
  The **Prototype** pattern is used to create a new object by copying an existing object.

Outline

Thanks!

Questions?

Repo: *https://github.com/EngAndres/ud-public/tree/main/courses/software-modeling*