Creational Patterns

Lect. PhD. Arthur Molnar

Babes-Bolyai University

arthur@cs.ubbcluj.ro

Overview

Creational Patterns

- Intro & Example
- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

Intro

- Creational patterns abstract the instantiation process
- They hide the object creation process (*"new is glue"*)
- Emphasis placed on fundamental behaviours that can be combined into complex ones
- *What* is created can be decided at compile, or at run time

A-Mazeing

- We use the common example of building a maze to study these patterns
- A maze is a set of connected *rooms*; each room knows its direct neighbours - another **room**, a **wall** or a **door** leading to another room
- Keeping it simple - directions are north, south, east, west
- Keeping it even simpler - ignore everything else (no multiplayer or pew pew ☺)

A-Mazeing

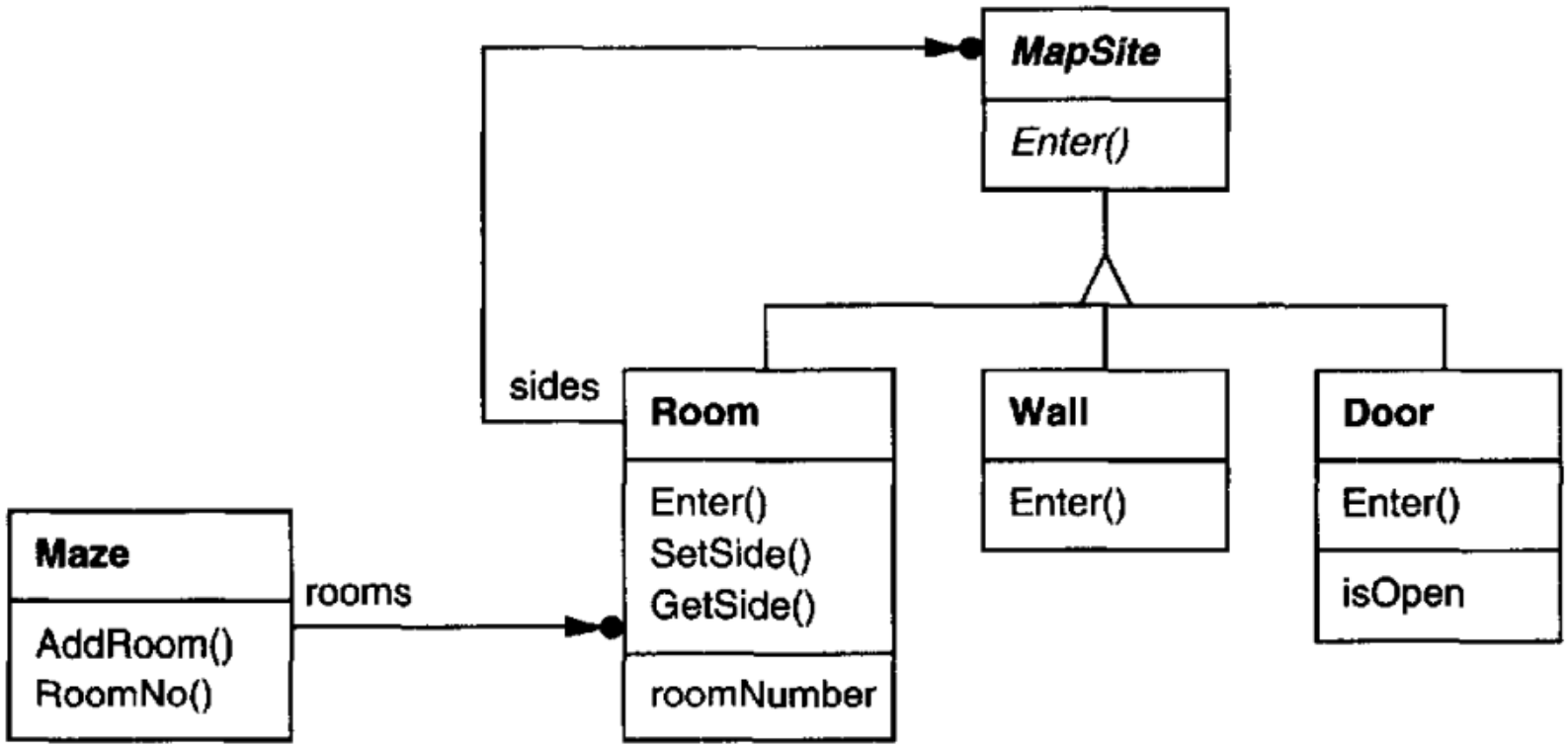


Figure: From [1]

A-Mazeing

- *MapSite* is abstract
- The rest are concrete, but can be *subclassed*
- Behaviour of *Enter()* depends on trying a door or another room

A-Mazeing

Maze source code

git: /ubb/dp/creational

A-Mazeing

What does the source code look like?

- It's incomplete - obviously
- Could move wall creation code to *Room* constructor...
- Big problem: **inflexible**
- We can't change maze layout, or the type of its elements (e.g. *PewPewRoom*) without changing the code
- Overriding it = new implementation, twice the work
- Changing in place = we might need this version too, kind of error-prone

A-Mazeing

Where creational patterns come in:

- Make the design more flexible (not *necessarily* smaller)
- Changing the maze type, or its elements should be easier (some doors need a key or spell, rooms might have a ticking bomb...)
- This can be achieved when we no longer hard-code the creation of the maze's elements
- *New is glue*, remember?

A-Mazeing

Where creational patterns come in

- **Factory Method:** instead of using constructor calls, call some virtual functions, which can be replaced by subclassing
- **Abstract Factory:** pass *CreateMaze* a parameter that can be used to create maze elements; you can change maze element types by providing a different parameter

A-Mazeing

Where creational patterns come in

- **Builder:** pass *CreateMaze* an object that can build an entire maze by creating rooms, walls and door sequentially, and then varying this using inheritance.
- **Prototype:** parameterize *CreateMaze* using prototypes for maze elements, which you clone, configure and add to the maze.
- **Singleton:** use it to ensure the maze program uses a single maze, to which all components have access.

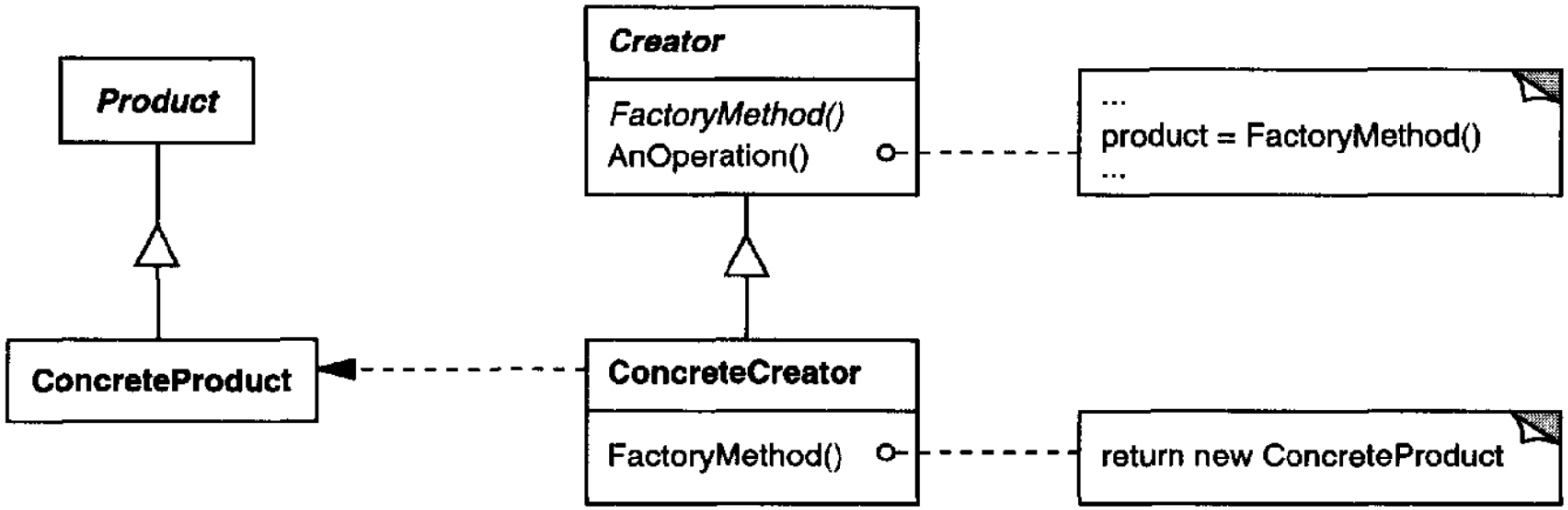
Factory Method

Gang of Four

"Define an interface for creating an object, but let subclasses decide which class to instantiate. The Factory method lets a class defer instantiation it uses to subclasses."

- Define an interface for object instantiation, but let subclassess decide the type that is created
- The *abstract factory* is one of this pattern's users

Factory Method



Factory Method

Varieties

- Creator can be concrete and create a default object, which subclasses can override (provides a hook)
- Creator can be parameterized, with the created type depending on the provided parameter (**e.g.** Logger.getLogger(String name) (concrete factory))
- Concrete creators can also be templated, according to the *Product* they should create

Factory

Source code

git: /src/ubb/dp/creational/FactoryMethodMaze

Abstract Factory

Intent

Provide an interface to create a family of related or dependent objects, without specifying concrete classes

Abstract Factory

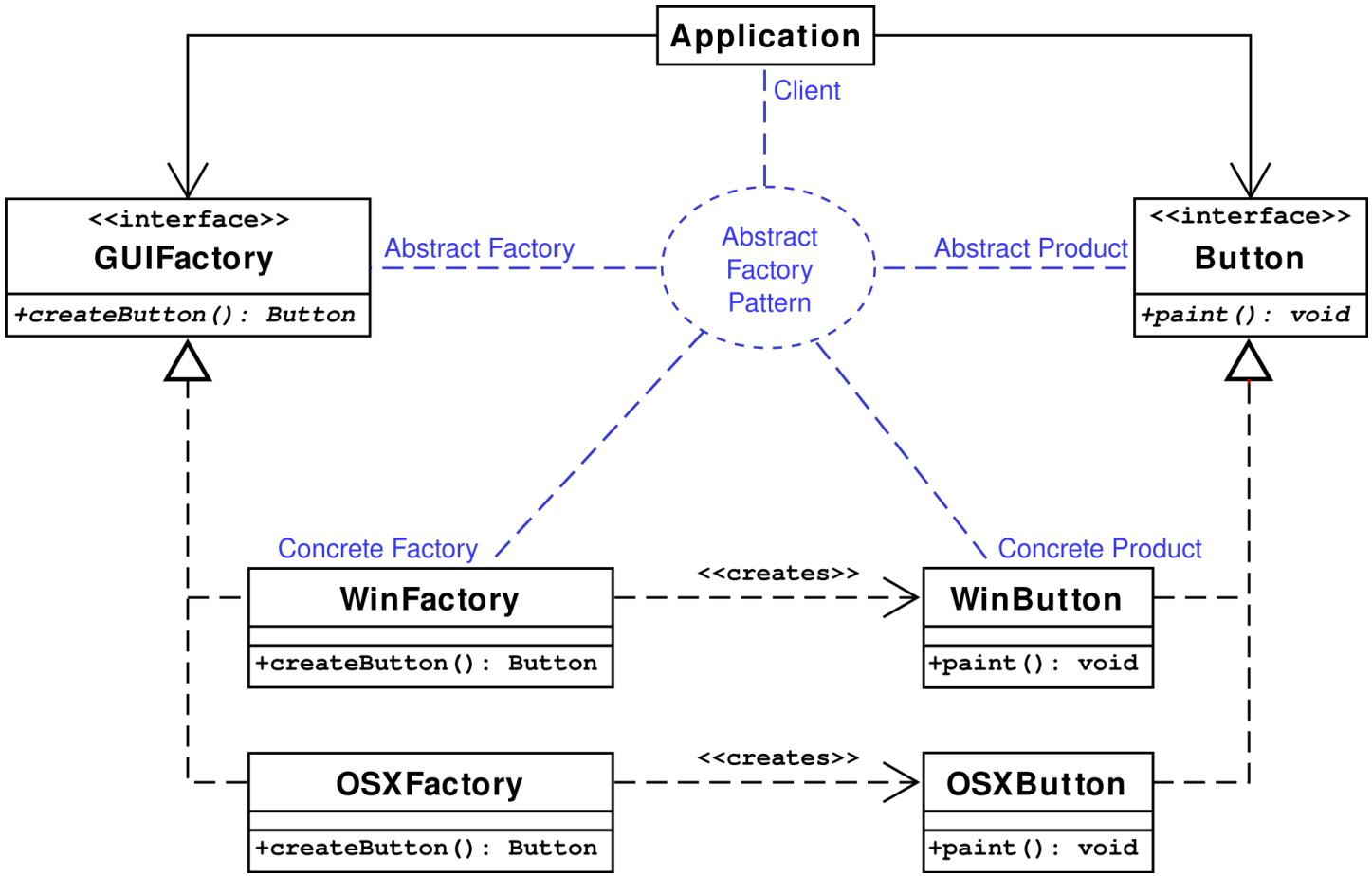


Figure: From https://en.wikipedia.org/wiki/Abstract_factory_pattern

Abstract Factory

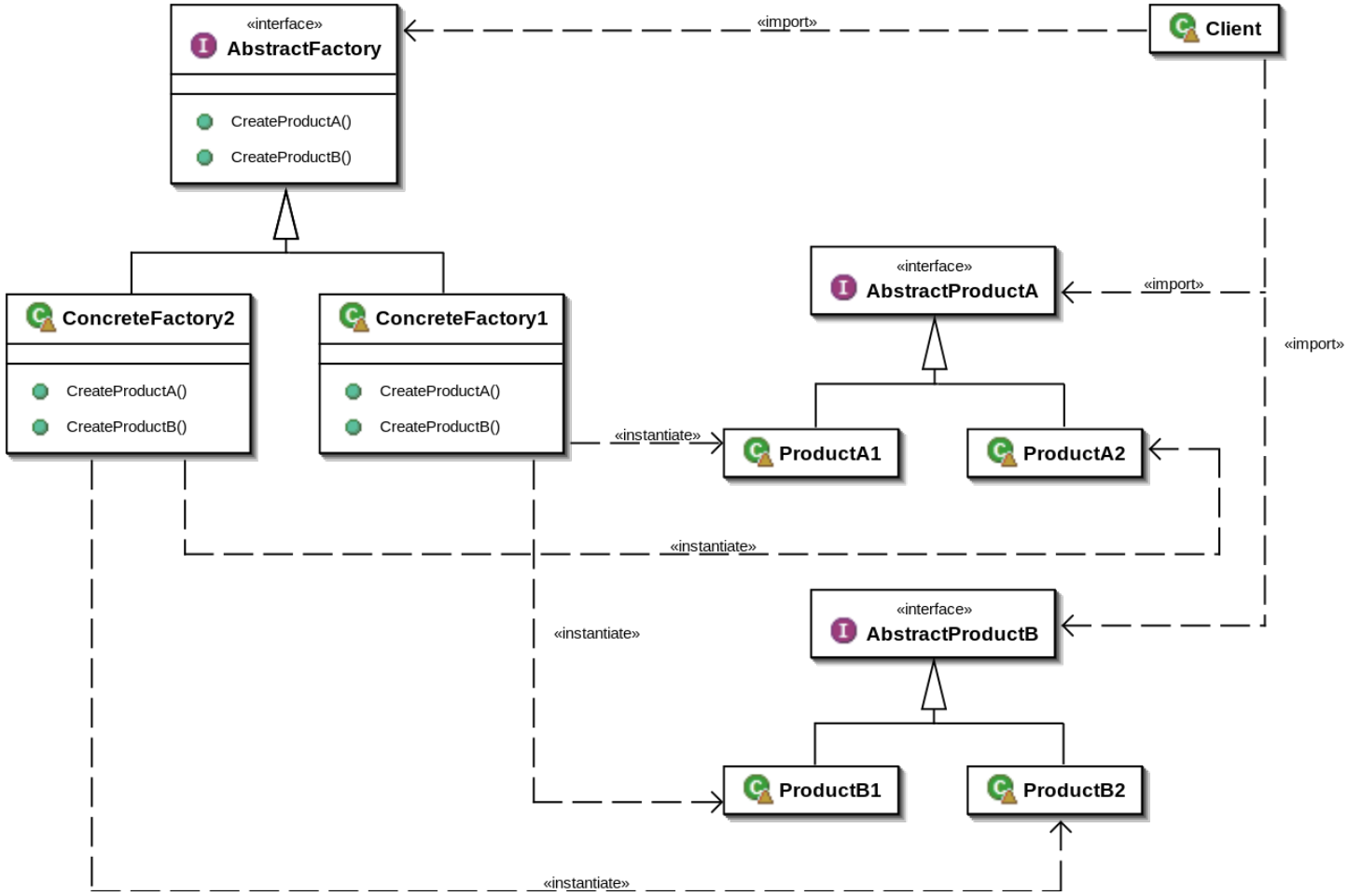


Figure: From https://en.wikipedia.org/wiki/Abstract_factory_pattern

Abstract Factory

- The base class is abstract, and several concrete factories implement it
- The *Client* only refers the *AbstractFactory* and abstract products, in order to remove dependency to actual implementations
- *Client* only commits to the interface, not the implementation $=>$ Open/Closed principle using polymorphism
- The *Factory* enforces dependency between classes (e.g. don't try to use a *macOS* button in a Windows context)

Abstract Factory

When to use, tips & tricks

- System independent from how products are created and composed
- System can be configured with several product families (hint: check out *Factory* if it's not a product family)
- Products should be used together
- Supporting new products is difficult, due to extensively specified interfaces
- Implementations are usually based on the *Factory Method*, and since there is usually a single concrete factory, it can be implemented as a *Singleton*

Abstract Factory

Source code

git: /src/ubb/dp/creational/AbstractFactoryMazeGame

Builder

- Separate the construction of a complex object from its representation
- The same construction process can result in different representations
- Construction is made step by step

Builder

Example: Rich Text Format (RTF) conversion

- "Understanding" the file format, **and** converting it to another one (e.g. ASCII, TeX, PDF representation) are different things
- The first part is common, the second one isn't
- A good solution will not duplicate code, and will be *open for extension* - additional converters can be added later on

Builder

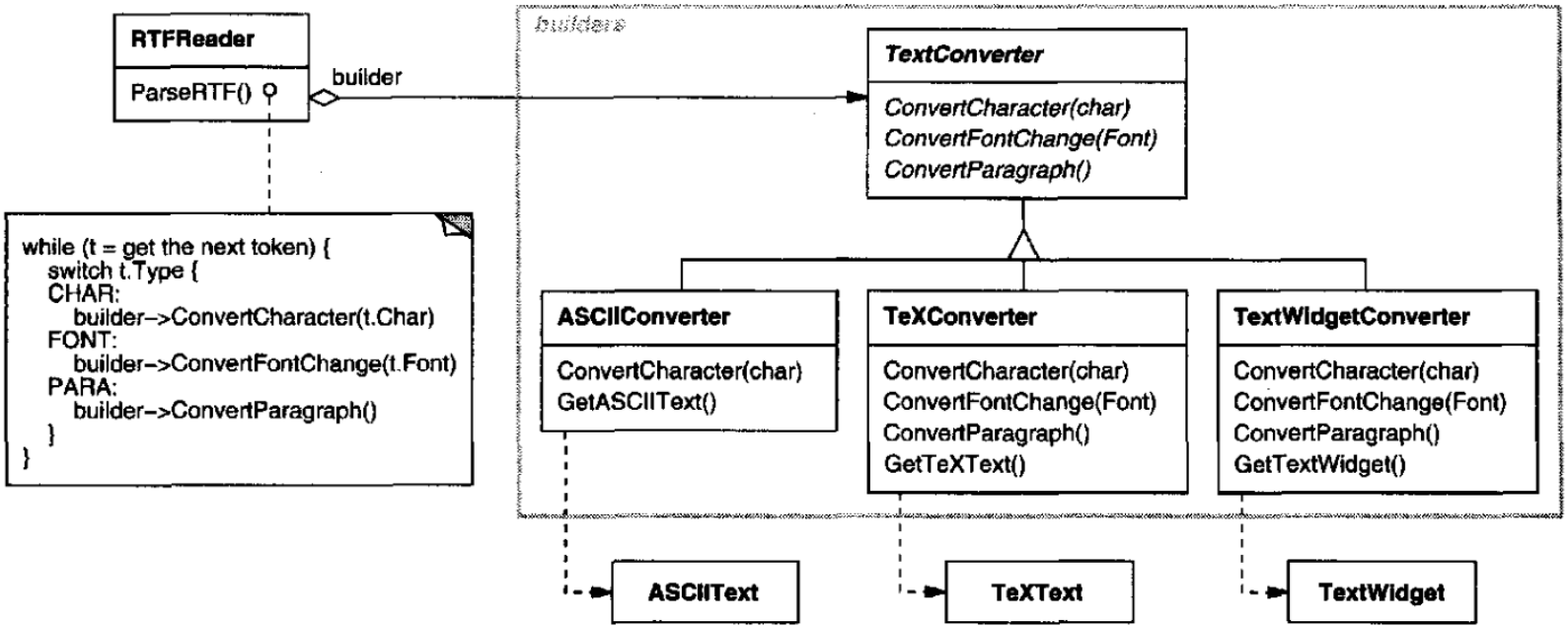


Figure: From [1]

Builder

- ASCII converter only cares about the text elements
- TeX converts all RTF elements to TeX
- The *TextWidget* converter produces a GUI element

Builder

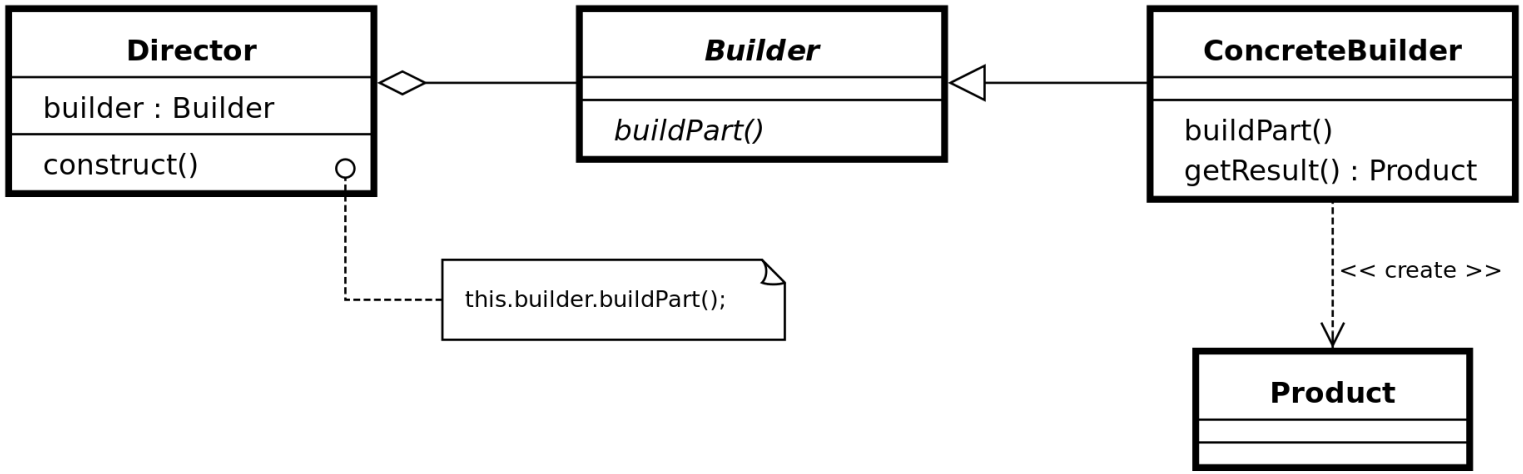


Figure: From `https://en.wikipedia.org/wiki/Builder_pattern`

Builder

Roles in the *Builder* pattern

- **Director:** constructs the object using the *Builder* interface (e.g. the RTF reader in our example)
- **Builder:** specifies the interface for creating the parts of the product (e.g. the *TextConverter*)
- **Concrete builder:** constructs & assembles the product, keeps track of the representation, provides an interface to retrieve it (e.g. the *TeXConverter* from our example)
- **Product:** the complex object being constructed (e.g. *ASCIIText*, *TextWidget*)

Builder

Using the builder pattern

- Lets you vary the product internal representation
- Improves encapsulation by hiding product internal representation
- Compared to *factory* patterns, provides more control over the build process

Builder

Implementation issues

- Sometimes appending the latest element to the *Product* under construction is enough (e.g. converters example, Java's *StringBuilder*, *Calendar*, *Locale*), but sometimes it isn't (e.g. the Maze game example), and additional data structures are required (e.g. parse trees)
- No abstract product class!?
- Base class with concrete but empty methods. Why? ☺

Builder

Source code

git: /src/ubb/dp/creational/BuilderMazeGame

Prototype

What is it?

- Use a prototype instance to decide the types of objects to create
- Clone the prototype to obtain new instances

When to use it?

- System independent of how product are created, composed and represented, **and**
- classes to instantiate are provided at run-time (dynamically loaded)
- want to avoid building a factory hierarchy

Prototype Example

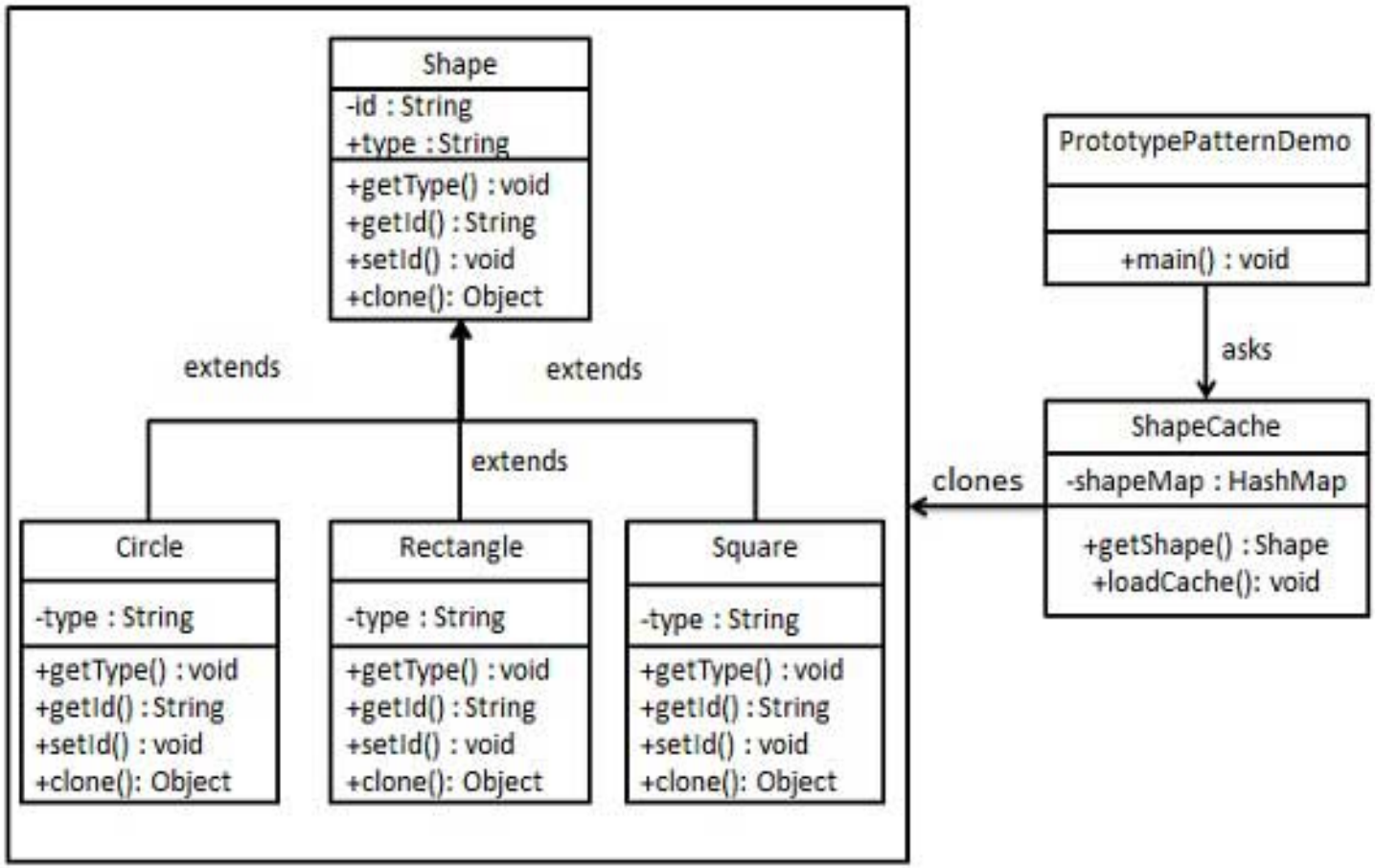


Figure: From `https://www.tutorialspoint.com/design_pattern/prototype_pattern.htm`

Prototype Example 1

Source code

git: /src/ubb/dp/creational/PrototypeShapes

Prototype

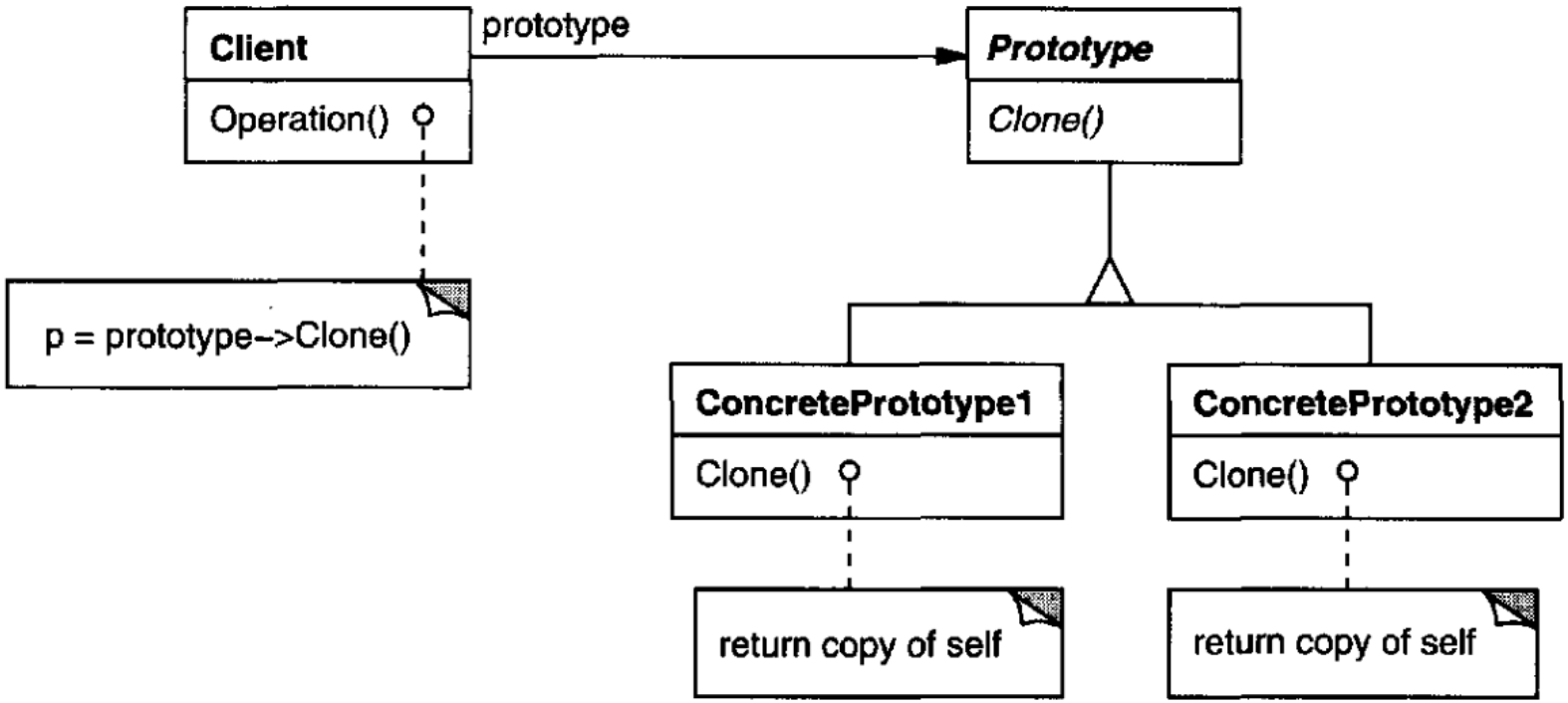


Figure: From [1]

Prototype

Pattern roles

- **Prototype:** declares the interface for cloning itself (e.g. could be Java's **Cloneable**, but read this first: `https://www.artima.com/intv/bloch13.html`
- **ConcretePrototype:** implements the actual operation to clone itself
- **Client:** creates new objects using cloning

Prototype Example 2

Building a maze using the *Prototype* pattern

Source code

git: /src/ubb/dp/creational/PrototypeMaze

Prototype

Consequences

- Shares benefits with the *factory* patterns (hides product details from clients)
- Better support using dynamic binding
- Avoid building class hierarchies for purposes of design
- Implement a *PrototypeManager* for handling prototype instances (maybe as *Singleton*?)

Drawbacks

- Implementation issues with shallow vs. deep copy, composition, circular references
- Implementation woes given *Java*'s *Cloneable* interface (does not include the *clone()* method, objects are built using field-copy and not constructors, leading to possible invariant violations)

Singleton Pattern

- Ensure a class has a single instance
- Provide a global access point to that instance
- **e.g.** graphical user interface, file system, database connection
- Instead of using a global variable, make the class itself responsible of its sole instance
- The *Singleton* class can be inherited from

Singleton Pattern

- *Singleton* design allows you to control the number of instances
- Allows extension through polymorphism, unlike *C++* static methods
- Instead of using a global variable, make the class itself responsible of its sole instance

Singleton Pattern

- The *Singleton* class can be inherited from
- Thread safety should be ensured
- *Lazy* versus *eager* instance creation
- Using a *SingletonRegistry* provides a global management point for several singleton classes
- *Multiton* is the singleton pattern allowing several instances

Singletons !?

Building a correct singleton maze factory in *Java*

Source code

git: /src/ubb/dp/creational/MazeSingleton