Object-Oriented Programming

Design Pattern Factory Method

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

Factory Method

```
#include "SortTraits.hpp"
class Sort {};
class QSort : public Sort {};
class InsertionSort : public Sort {};

Sort* factory(int size) {
    if (size < SortTraits::Cutoff)
        return new InsertionSort();
    else
        return new QSort();
}
```

Factory Method

A class pattern that defines an interface for creating an object but lets subclasses decide which class to instantiate lets a class defer instantiation to subclasses.

- Class Creational Pattern

- AKA: Virtual Constructor

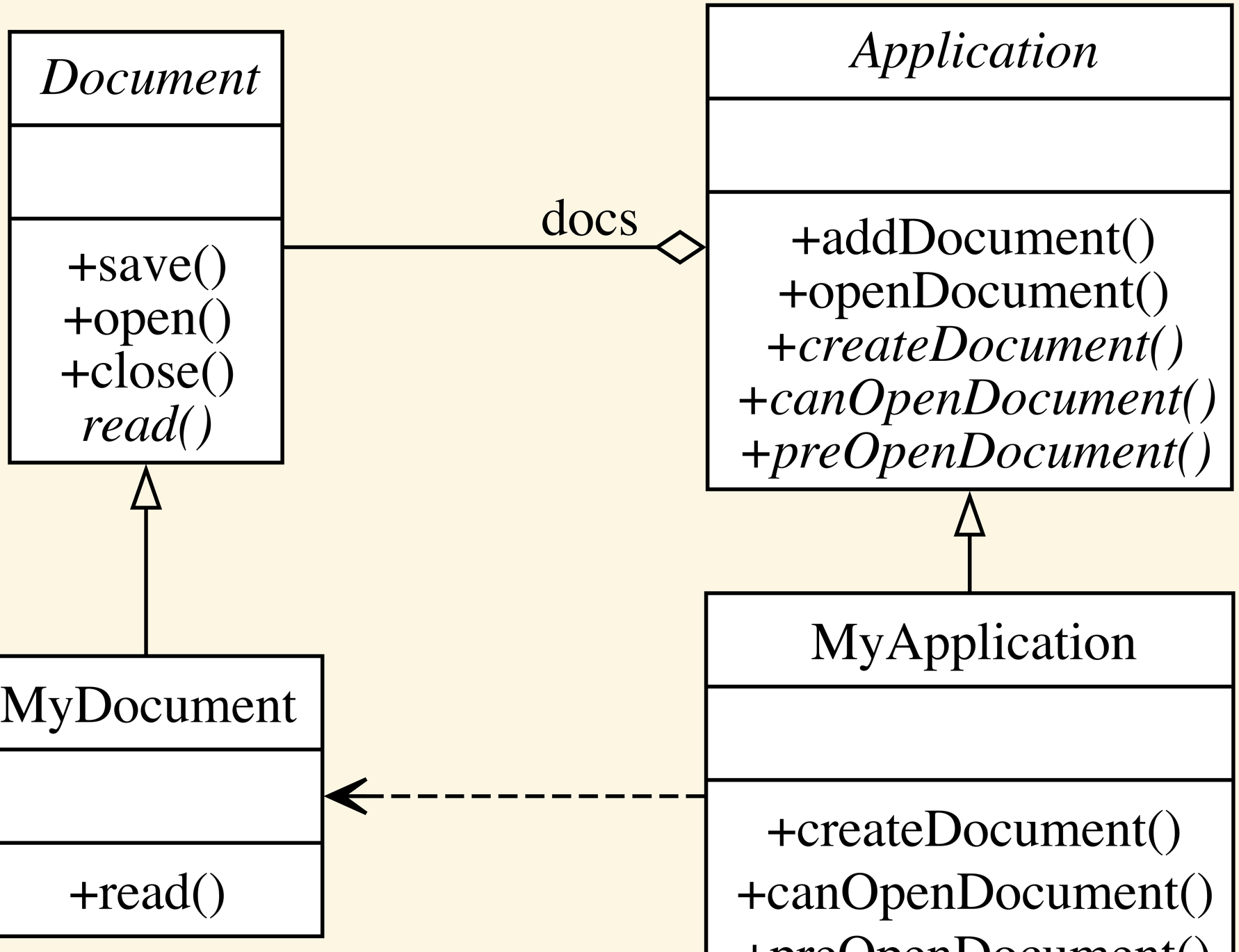
Factory Method

- Frameworks use abstract classes to define and maintain relationships between objects

- Also responsible for creating these objects

- However, knowledge of what type of object to create may not be in the framework

Factory Method: Motivation



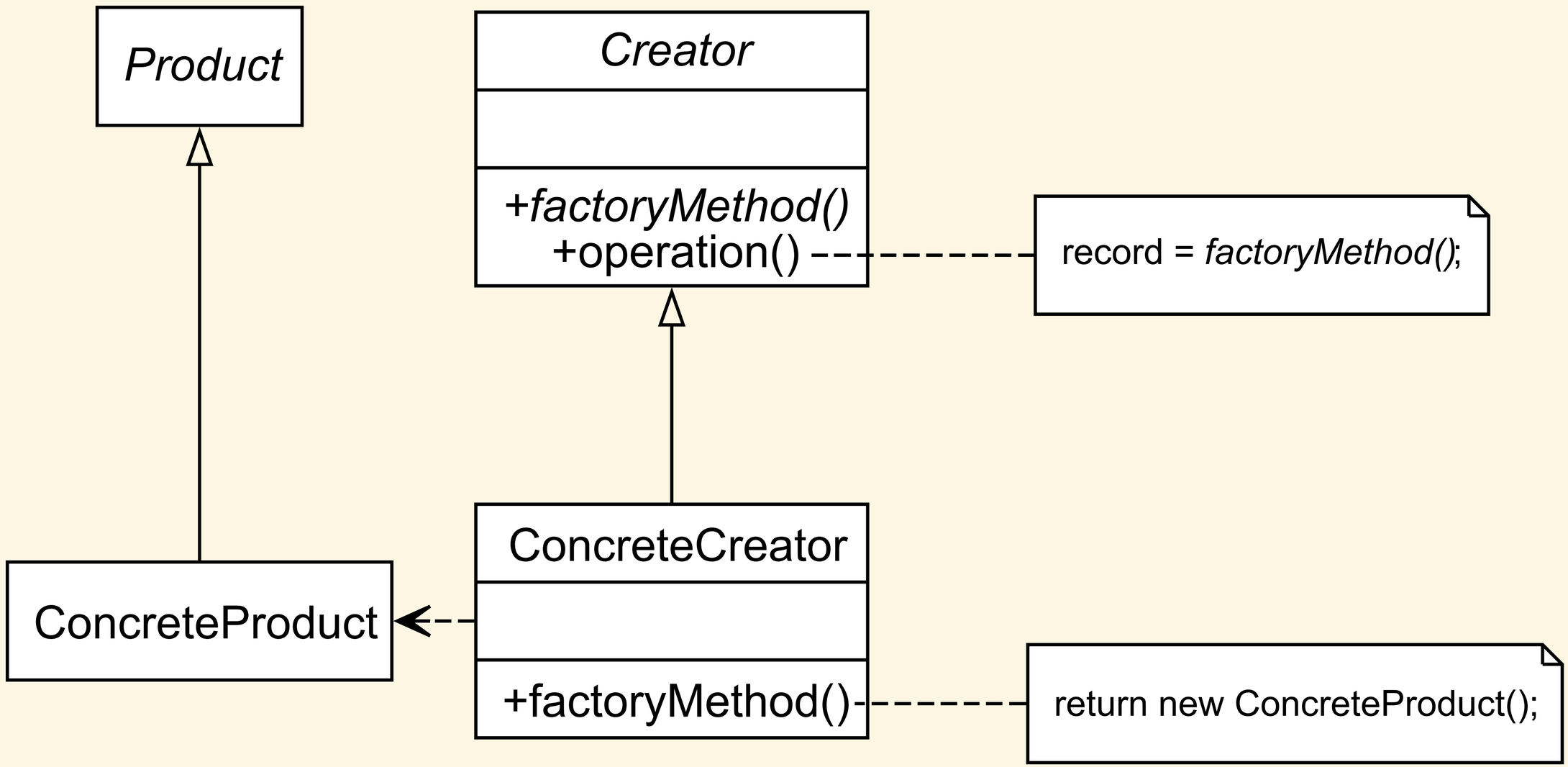
Factory Method: Applicability

- Classes can't anticipate the class of objects they must create

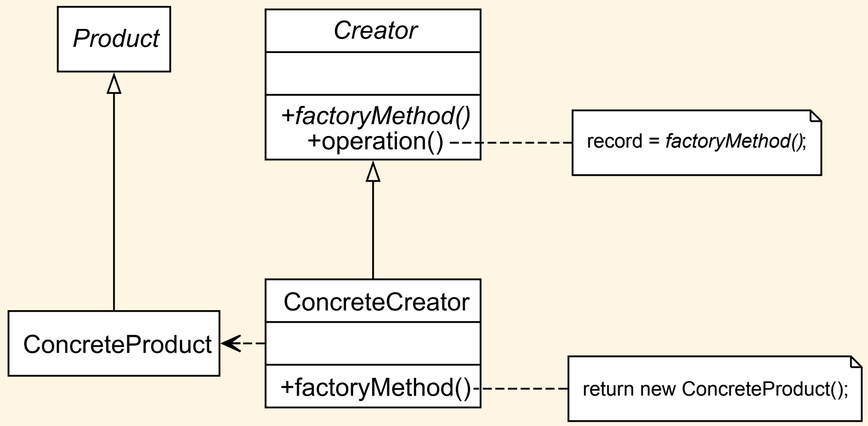
- Classes want subclasses to specify the objects they create

- Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate

Factory Method: Structure



Factory Method: Product Participants



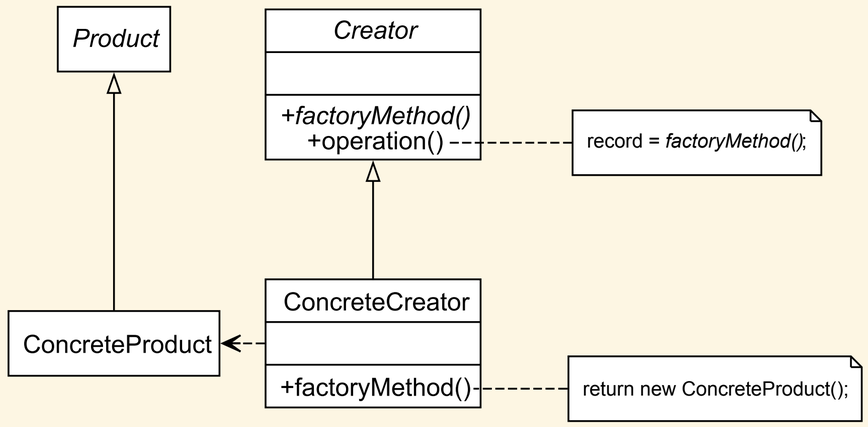
- `Product` (e.g., Document)

  Defines the interface of objects the *factory method* creates

- `ConcreteProduct` (e.g., MyDocument)

  Implements the Product interface

Factory Method: Creator Participants



- `Creator` (e.g., Application)

  Declares the *factory method*, which returns an object of type Product

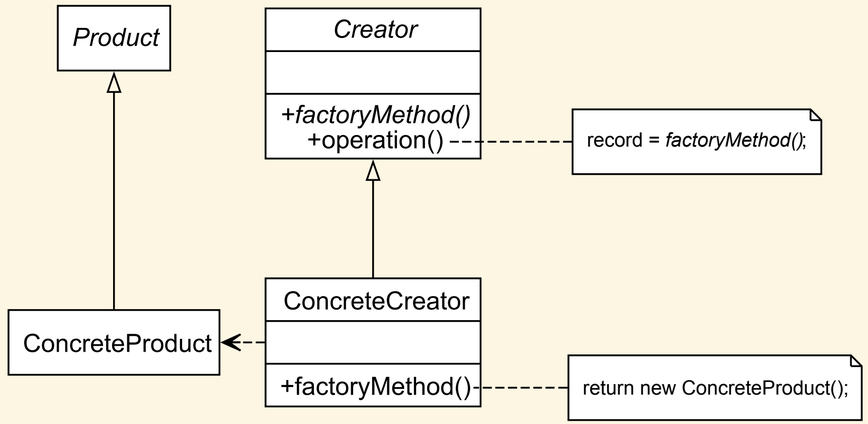
  May also define a default implementation of the factory method that returns a default ConcreteProduct object

  May call the *factory method* to create a Product object.

- `ConcreteCreator` (e.g., MyApplication)

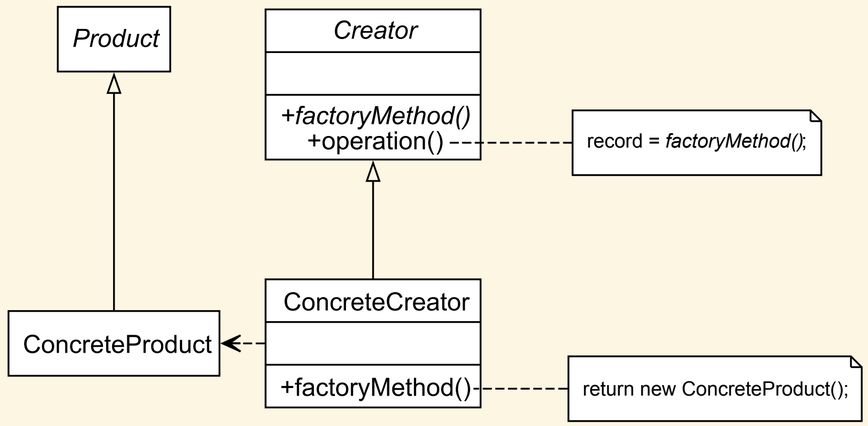
  Overrides the factory method to return an instance of a ConcreteProduct.

Factory Method: Collaborations



- `Creator` relies on subclasses to define the factory method so that it returns an instance of the appropriate `ConcreteProduct`

Factory Method: Consequences

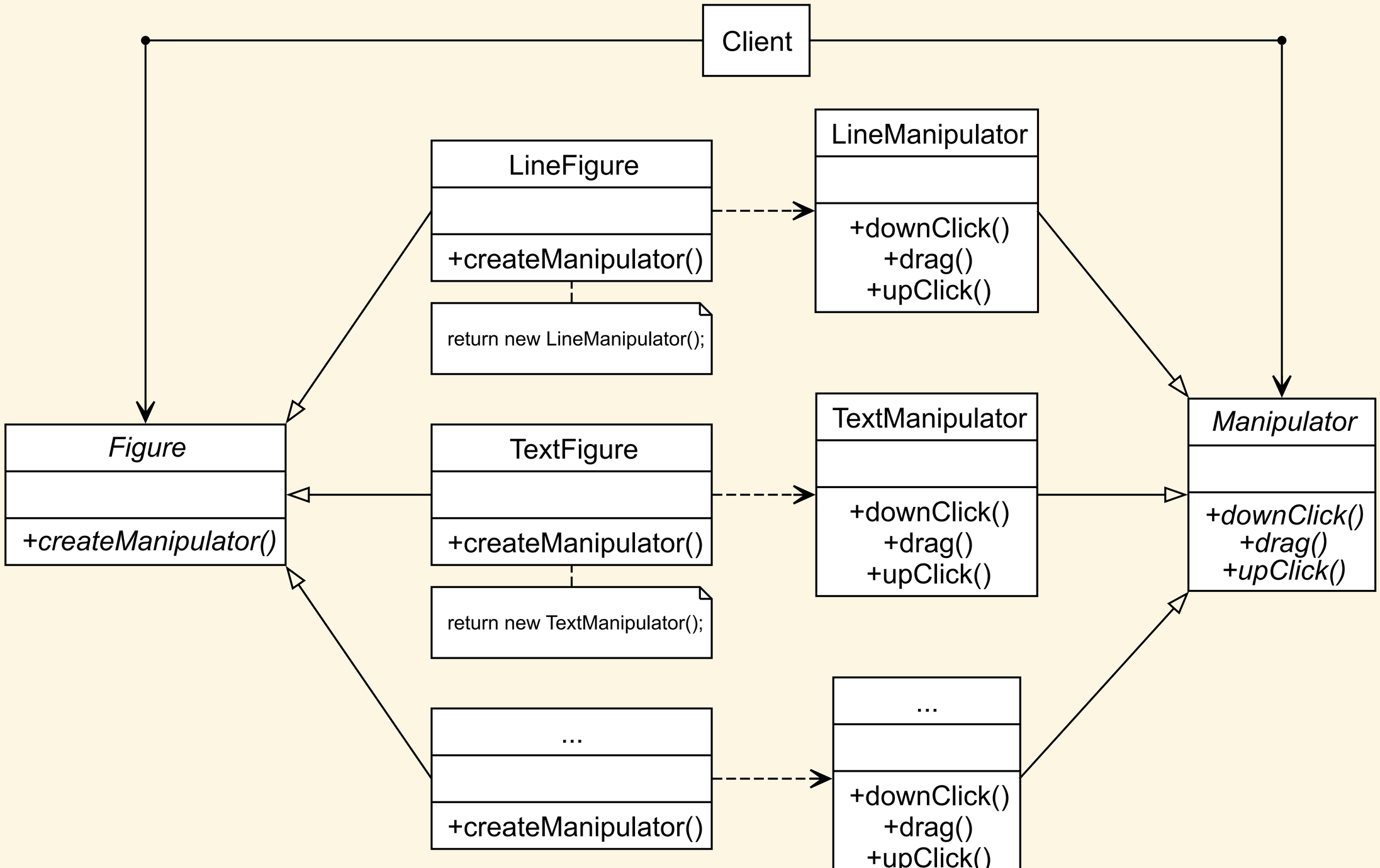


- Code only deals with `Product` interfaces so that it can work with any user-defined `ConcreteProduct` classes

- Does require a client to subclass `Creator` for each unique `ConcreteProduct`

- Can define a non-abstract default, e.g., `CreateFileDialog`, that subclasses can override if needed

Factory Method: Parallel Class Hierarchies



Implementation: Maze

```
Maze* MazeGame::createMaze() {
    Maze* aMaze = new Maze();

    Room* r1 = new Room(1);
    aMaze->AddRoom(r1);

    Room* r2 = new Room(2);
    aMaze->AddRoom(r2);

    Door* theDoor = new Door(r1, r2);

    r1->SetSide(North, new Wall());
    r1->SetSide(East, theDoor);
    r1->SetSide(South, new Wall());
    r1->SetSide(West, new Wall());

    r2->SetSide(North, new Wall());
    r2->SetSide(East, new Wall());
    r2->SetSide(South, new Wall());
    r2->SetSide(West, theDoor);

    return aMaze;
}
```

Implementation: Factory

```
class MazeGame {
public:

    // creates a new predefined maze
    Maze* createMaze();

    // create the appropriate maze
    // @pattern factory method
    virtual Maze* makeMaze() const {
        return new Maze;
    }

    // create the appropriate room
    // @pattern factory method
    virtual Room* makeRoom(int n) const {
        return new Room(n);
    }

    // create the appropriate wall
    // @pattern factory method
    virtual Wall* makeWall() const {
        return new Wall;
    }

    // create the appropriate door
    // @pattern factory method
    virtual Door* makeDoor(Room* r1, Room* r2) const {
        return new Door(r1, r2);
    }
};
```

Implementation: Using Factory

```
Maze* MazeGame::createMaze() {
    Maze* aMaze = makeMaze();

    Room* r1 = makeRoom(1);
    aMaze->AddRoom(r1);

    Room* r2 = makeRoom(2);
    aMaze->AddRoom(r2);

    Door* theDoor = makeDoor(r1, r2);

    r1->SetSide(North, makeWall());
    r1->SetSide(East, theDoor);
    r1->SetSide(South, makeWall());
    r1->SetSide(West, makeWall());

    r2->SetSide(North, makeWall());
    r2->SetSide(East, makeWall());
    r2->SetSide(South, makeWall());
    r2->SetSide(West, theDoor);

    return aMaze;
}
```

Implementation: Subclassing

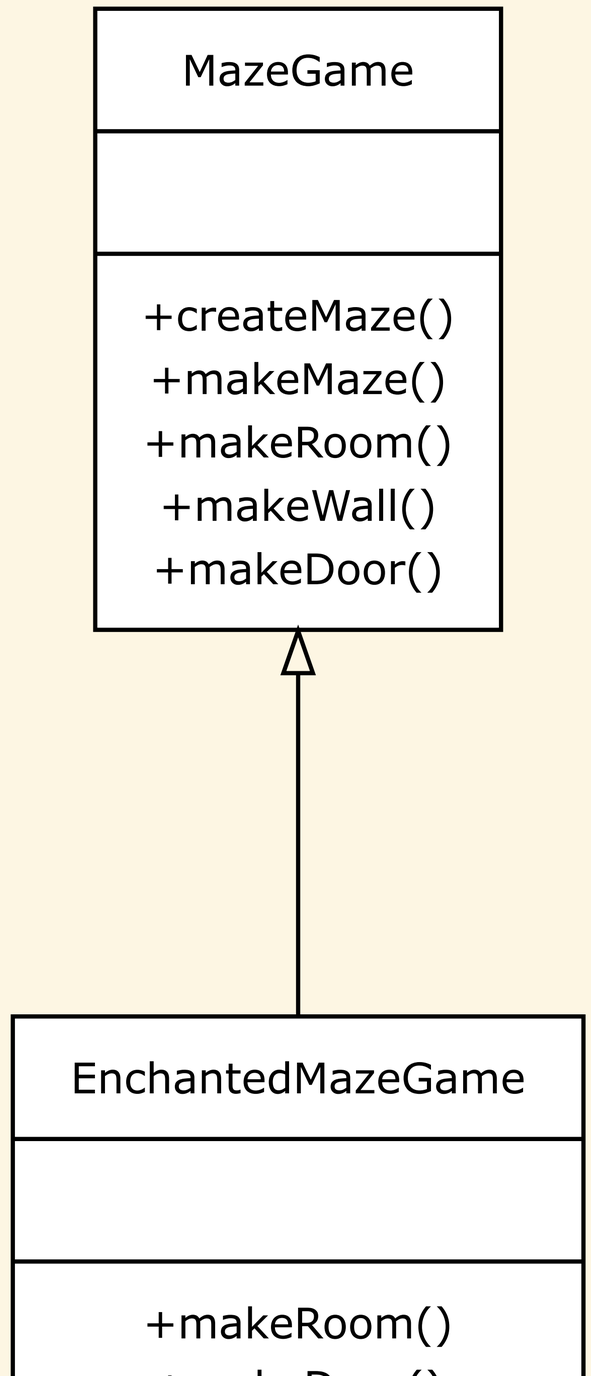
```
class EnchantedMazeGame : public MazeGame {
public:
    EnchantedMazeGame();

    // create an enchanted room
    // @pattern factory method
    Room* makeRoom(int n) const override {
        return new EnchantedRoom(n, castSpell());
    }

    // create a door requiring spells
    // @pattern factory method
    Door* makeDoor(Room* r1, Room* r2) const override {
        return new DoorNeedingSpell(r1, r2);
    }

protected:
    Spell* castSpell() const;
};
```

UML


MazeGame
+createMaze()
+makeMaze()
+makeRoom()
+makeWall()
+makeDoor()
EnchantedMazeGame
+makeRoom()

Lazy Initialization

```
class Creator {
public:

    // current product
    Product* getProduct() {

        // lazy initialization
        if (product == nullptr) {
            product = createProduct();
        }

        return product;
    }

protected:

    // create the appropriate product
    // @pattern factory method
    virtual Product* createProduct();

private:
    Product* product = nullptr;
};
```

Known Uses

Factory methods pervade toolkits and frameworks

Related Patterns

- *Abstract Factory*

  Often implemented with factory methods

- *Template Methods*

  Typically called to create a proper object