Creational Patterns

- control the process of *creating* new objects
- help making a system *independent* of how its objects are created, composed, or represented
- we will study the following ones:
  - **Singleton**
  - **Abstract Factory**
  - **Factory Method**
  - **Builder**

Structure of a Design Pattern

- **Intent**: what does it do? Why?
- **Motivation**: a concrete example or a scenario which requires an application of the pattern
- **Participants**: classes (concrete or abstract), the interfaces and objects which are used
- **Structure**: the generic description of the pattern, usually expressed in UML
- **Collaborations**: how different classes involved communicate with each other

Singleton pattern

Motivation: example

- almost every application exchanges data at run-time with a *database*
- for the application to execute one and only one transaction at run-time, we must be sure that at every moment we are manipulating the same session (in the database sense)

Singleton (2)

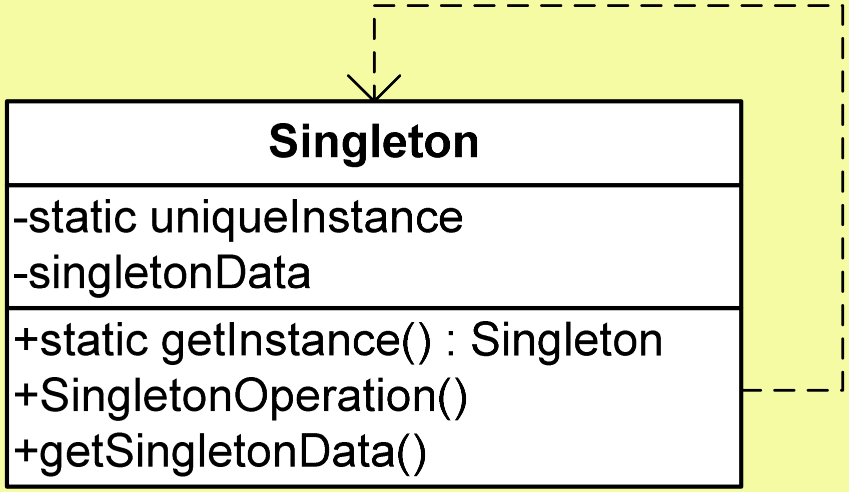
Intent

- ensure a class has only one instance,
- provide a global point of access to it
- any class which makes use of a Singleton always manipulates the same instance of it

Participants

- only one: the class responsible for creating its only instance.

Singleton: structure



Singleton (3)

Collaborations

Clients access a Singleton instance through:

- Singleton static **getInstance()** operation: returns the reference to the unique instance of the class
- The set of public methods defined for the Singleton object

Abstract Factory

Motivation: example

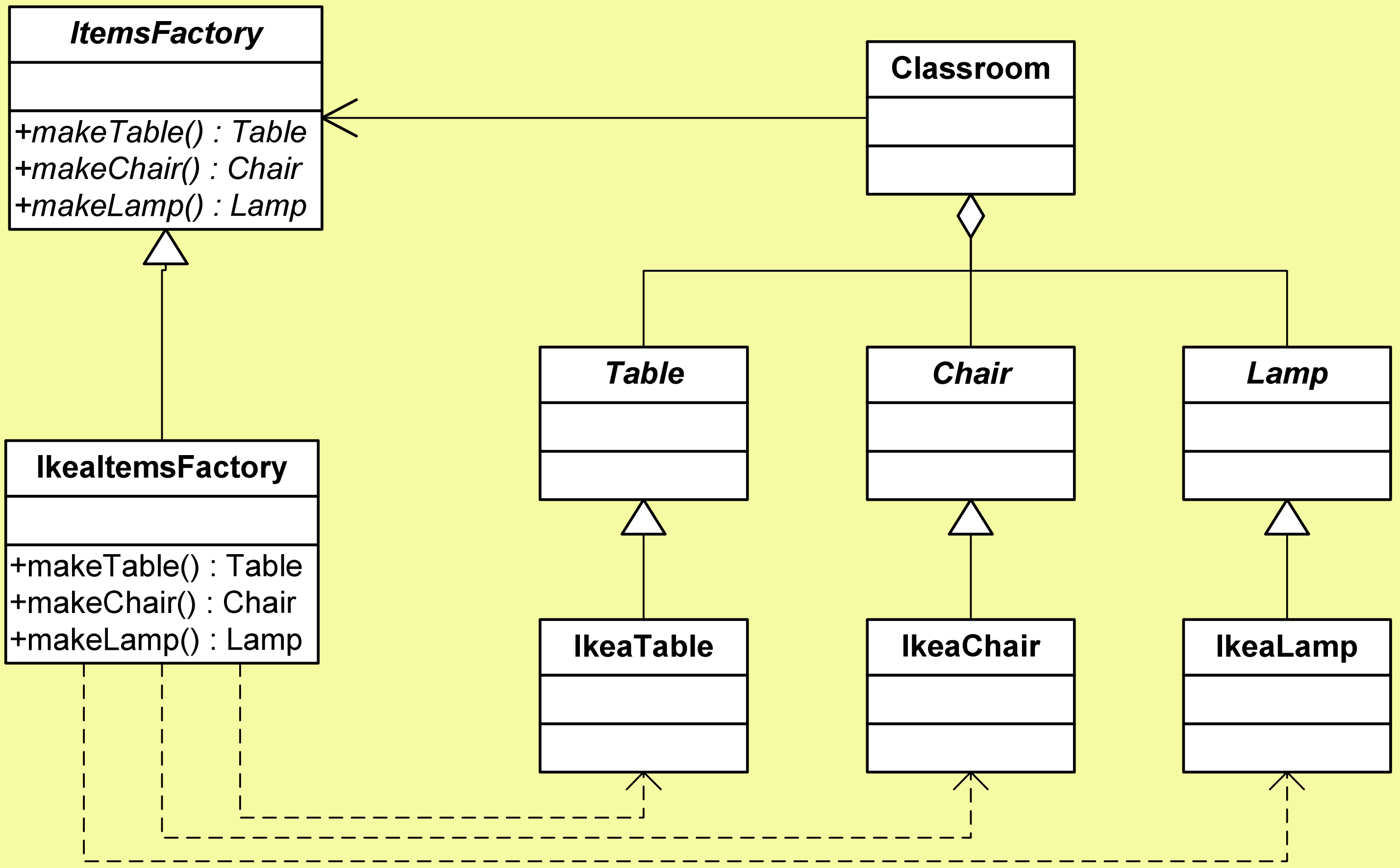
- Suppose we want to model the content of a classroom: tables, chairs and lamps
- The type of such elements can change from one room to another, or from one school to another
- Therefore, for any given room we must NOT hard code its specific elements. That would make it difficult to change later the type of the content.

Abstract Factory (2)

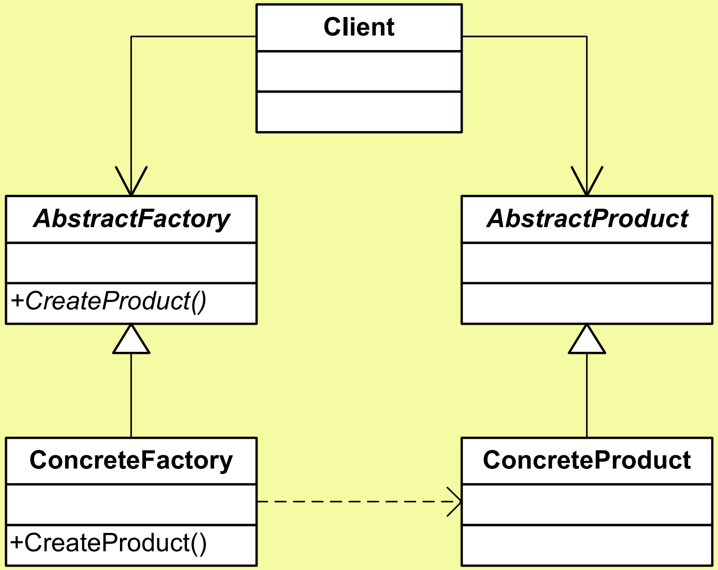
- **Intent**

  provide an interface to create a family of related objects without specifying their concrete classes

Abstract Factory: example implemented



Abstract Factory



Participants

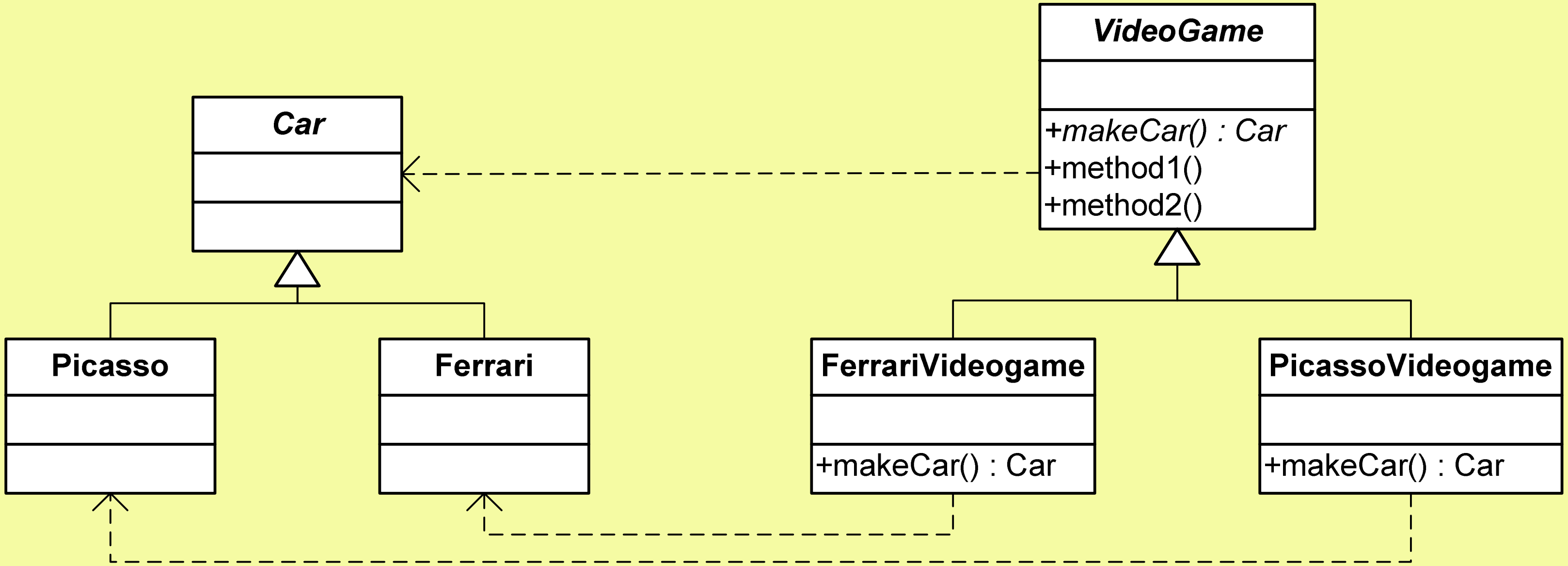
- AbstractFactory: declares an interface for operations that create abstract product objects
- ConcreteFactory: implements the operations to create concrete products
- AbstractProduct: declares an interface for a type of product objects
- ConcreteProduct: implements the AbstractProduct interface
- Client: only uses interfaces declared by *abstract classes*

Pattern: Factory Method

Motivation: example

- – Suppose we want to design a racing cars videogame: we want to be able to add new cars independently from the game design.
- – The videogame must know the cars’ interface but not the type of implemented cars.
- – Therefore: for the game to make use of a car, it must be able to create a car without knowing its type!!!

Factory Method: example implemented



Factory Method Pattern (2)

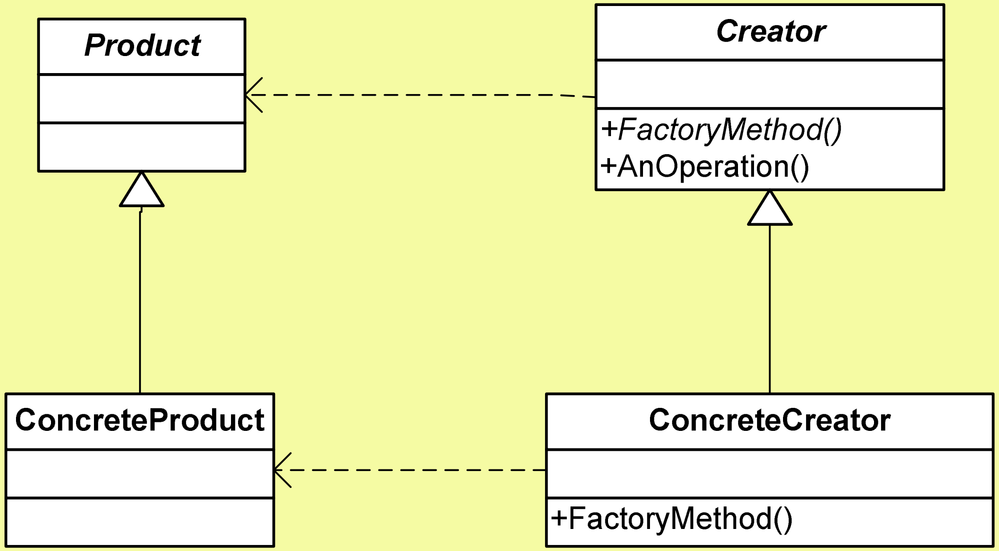
- **Intent**
  - Définir une classe (abstraite) qui implémente toutes ses méthodes qui utilisent des objets et ne fait que déclarer les méthodes de fabrication des objets utilisés.
  - La méthode de fabrication permet à la classe abstraite de déléguer l’instanciation des objets utilisés à d’autres classes.

Factory Method Pattern

Intent

- define an interface for creating an object,but
- let subclasses decide which class to instantiate
- Factory Method lets a class defer instantiation to subclasses

Factory Method Structure



- Product: defines the interface of the objects which are produced by the factory method
- ConcreteProduct: implements the Product interface
- Creator: declares the factory method and implements other methods
- ConcreteCreator: overrides the factory method to return an instance of ConcreteProduct

Creational Patterns: putting it all together

Example :

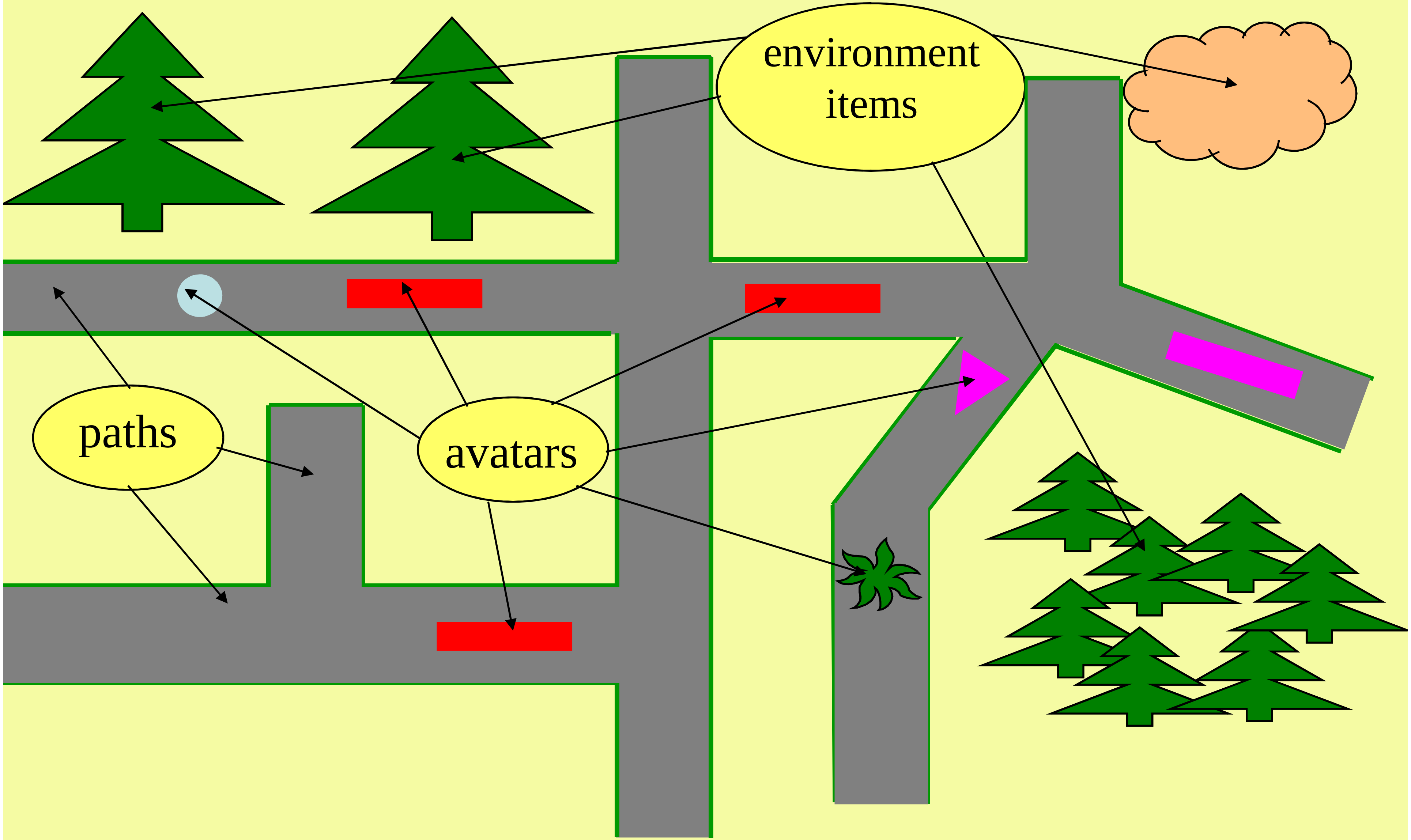
- We want to design an application to model living creatures (avatars) moving along paths.
- Paths are located in an environment which includes other items besides paths.

Putting it all together (2)

- We can build spaces made of
  - segments of paths
  - crossings
  - an environment (trees, rivers,…)

- At any time, we want to be able to create new avatars and put them on one of the paths.

Putting it all together (3)



Putting it all together (4)

The Animation class uses the following objects:

- spaces and their environment items
- segments of paths
- crossings
- avatars

Putting it all together (5)

1. We create an Abstract Factory

```
interface IAnimationFactory
    EnvironmentItem makeEnvironmentItem(…)
    Space makeSpace(…)
    Crossing makeCrossing(…)
    PathSegment makePathSegment(…)
    Avatar makeAvatar(…)
```

Putting it all together (6)

2. For every object that we manipulate (space, environment item, avatar, path, crossing), we define its usage interface inside Animation class
   - ISpace
   - IEnvironmentItem
   - IPath
   - ICrossing
   - IAvatar

Putting it all together (7)

(1-2) allow us to create Animation class without knowing the concrete type of the objects it manipulates

Putting it all together (8)

Enter Factory Method:

3. Animation class is an abstract class which
   - declares an abstract method which returns a factory through a reference to IAnimationFactory
   - Implements all the methods of the application which uses, through their interfaces, the constructed objects
4. Eventually, we create a concrete class ConcreteAnimation which inherits from Animation and overrides the method to create a Factory

Putting it all together (9)

```
/**
This class implements the whole application except for the concrete object instantiation mechanism,
which is delegated to a subclass
*/
abstract class Animation {
        private ISpace space;

        public IAnimationFactory makeAnimationFactory();

        void run(…) {
          space = makeAnimationFactory().makeSpace();
          …
          IAvatar av = makeAnimationFactory().makeAvatar(…);
          space.addAvatar(av);
          …
          imethod(…);
        }

        void imethod(…) { space.addPath(makeAnimationFactory().makePath(…));
          …
        };}
```

Putting it all together (10)

```
// A concrete Animation class
class RuralAnimation extends Animation {

    IAnimationFactory makeAnimationFactory(…) {
      return RuralAnimationFactory.getInstance(…);
    }

    public static void main (String [] args) {
      RuralAnimation ra = new RuralAnimation(…);
      ra.run(…);
    }
}
```

Putting it all together (11)

5. RuralAnimationFactory can be implemented as a Singleton in order to control the whole set of objects by means of a unique factory object

Summing up…

In every application, we can systematically make use of

- Abstract Factory pattern to provide the application class with a interface to create the objects used by the application
- Factory Method pattern to implement the creation of the objects factory. This way, we hide the concrete type of the objects factory to the application class.
- Singleton pattern to guarantee that the same and only factory is used by the application.

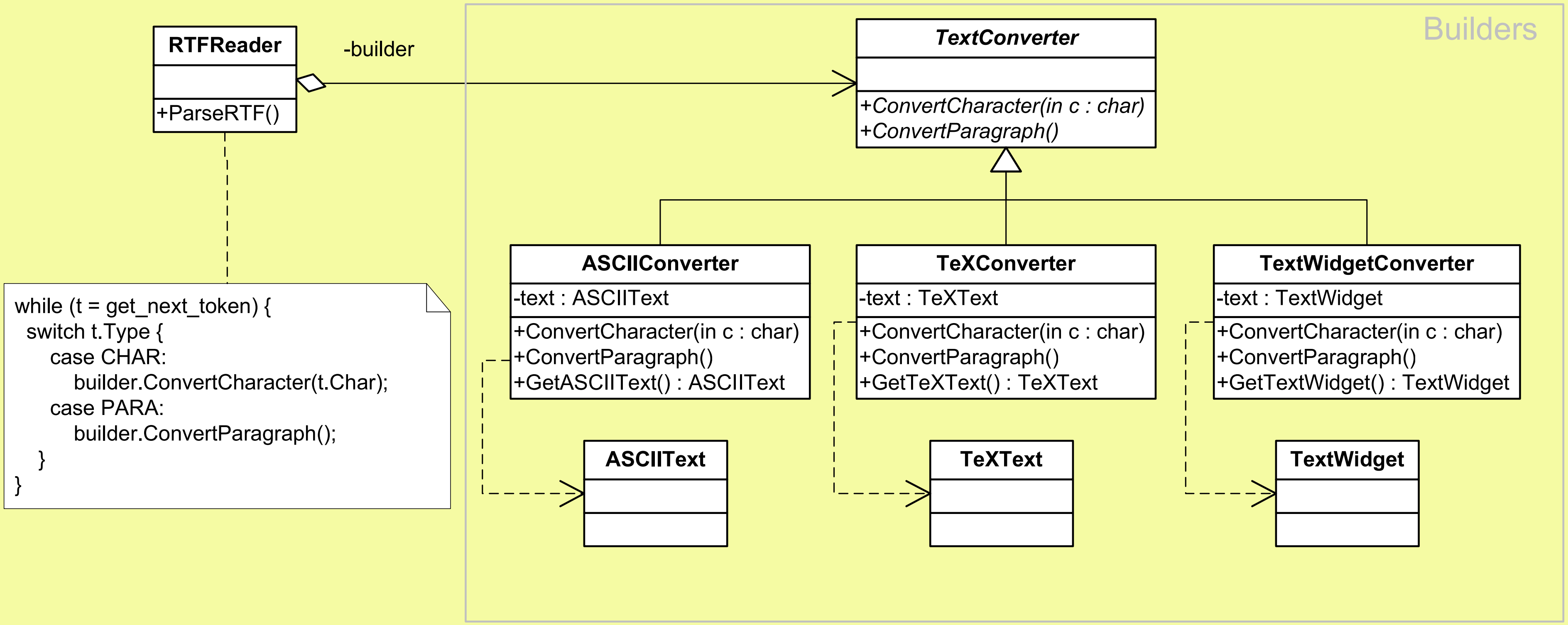
Builder Pattern: Motivation

- A reader for RTF (Rich Text Format) documents format should be able to convert RTF to many text formats
- Problem: the number of possible conversions is open-ended (ASCII, TeX, PDF, …)
- How can we design the reader application so that we can add a new conversion without modifying the reader?

Builder: Intent

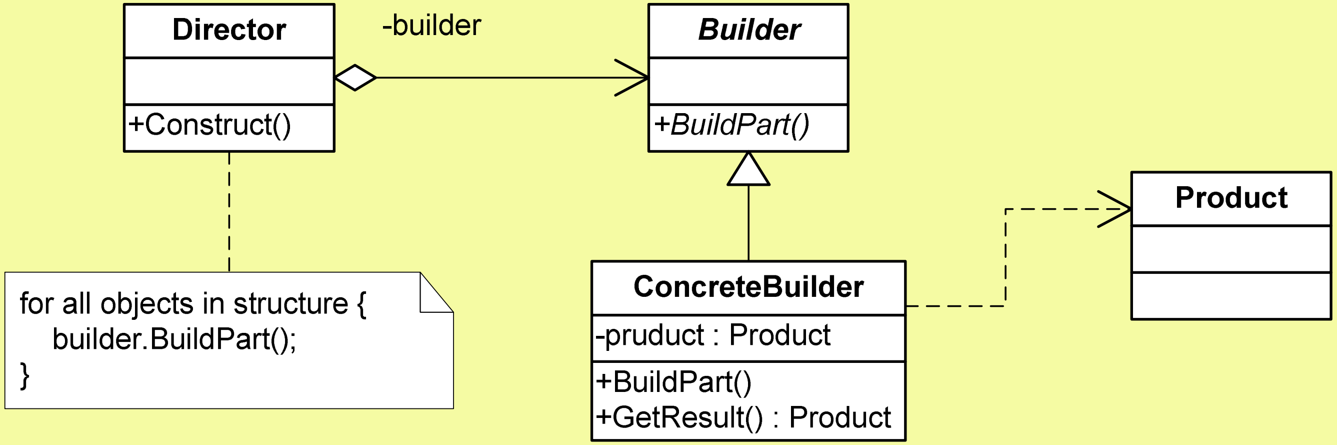
Separate the construction of a complex object from its representation so that the same construction process can create different representations

Builder Pattern: Motivation (2)



```
while (t = get_next_token) {
  switch t.Type {
      case CHAR:
          builder.ConvertCharacter(t.Char);
      case PARA:
          builder.ConvertParagraph();
    }
}
```

Builder: Structure & Participants



- Builder
  - Specifies an abstract interface for creating parts of a product object
- ConcreteBuilder
  - constructs the product by implementing the Builder interface
  - defines and keeps track of the representation it creates
- Director
  - constructs an object using the Builder interface
- Product
  - represents the complex object under construction
  - Includes classes that define the constituent parts, including interfaces for assembling the parts into the final result

Builder: Consequences

- It lets you vary a product’s internal representation
- It isolates code for construction and representation
- It gives you a finer control over the construction process