Tecniche di Progettazione: Design Patterns

GoF: Builder, Chain Of Responsibility, Flyweight

Builder: intent

- As in GoF
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations
- In simpler worlds
  - Separate the logic of the construction of a complex object (algorithm, recipie…) from the construction of the single pieces and their assembly.
  - Using delegation instead of inheritance as in Factory method.
  - Products can be different from each other

Builder: structure

Director

+construct():void

1

Builder

+buildPart():void

for all objects in structure {
builder.buildPart();
}

ConcreteBuilder

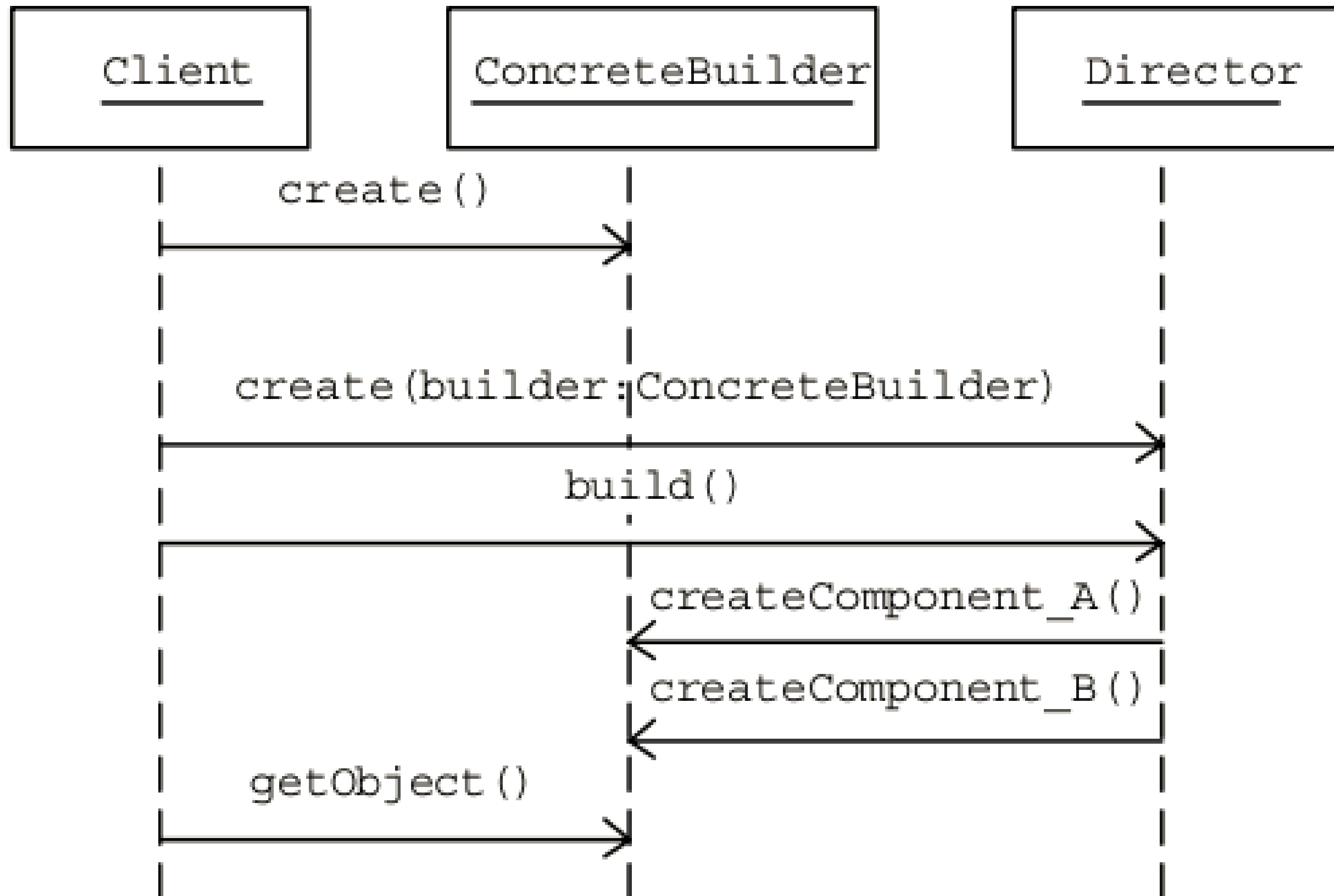
+buildPart():void
+getResult():void

<<creates>>

Product

Builder: participants

- Builder
  - Specifies an interface for creating/assembling parts of a product.
- ConcreteBuilder
  - Implements the Builder interface
- Director
  - Constructs an object of an unknown type using the Builder interface
- Product
  - Complete object returned by invoking getResult() on the ConcreteBuilder


Client
ConcreteBuilder
Director
create()
create(builder:ConcreteBuilder)
build()
createComponent_A()
createComponent_B()
getObject()

Builder: consequences

- Isolates code for construction and representation
  - Construction logic is encapsulated within the director
  - Product structure is encapsulated within the concrete builder
  - => Lets you vary a product's internal representation
- Supports fine control over the construction process
  - Breaks the process into small steps

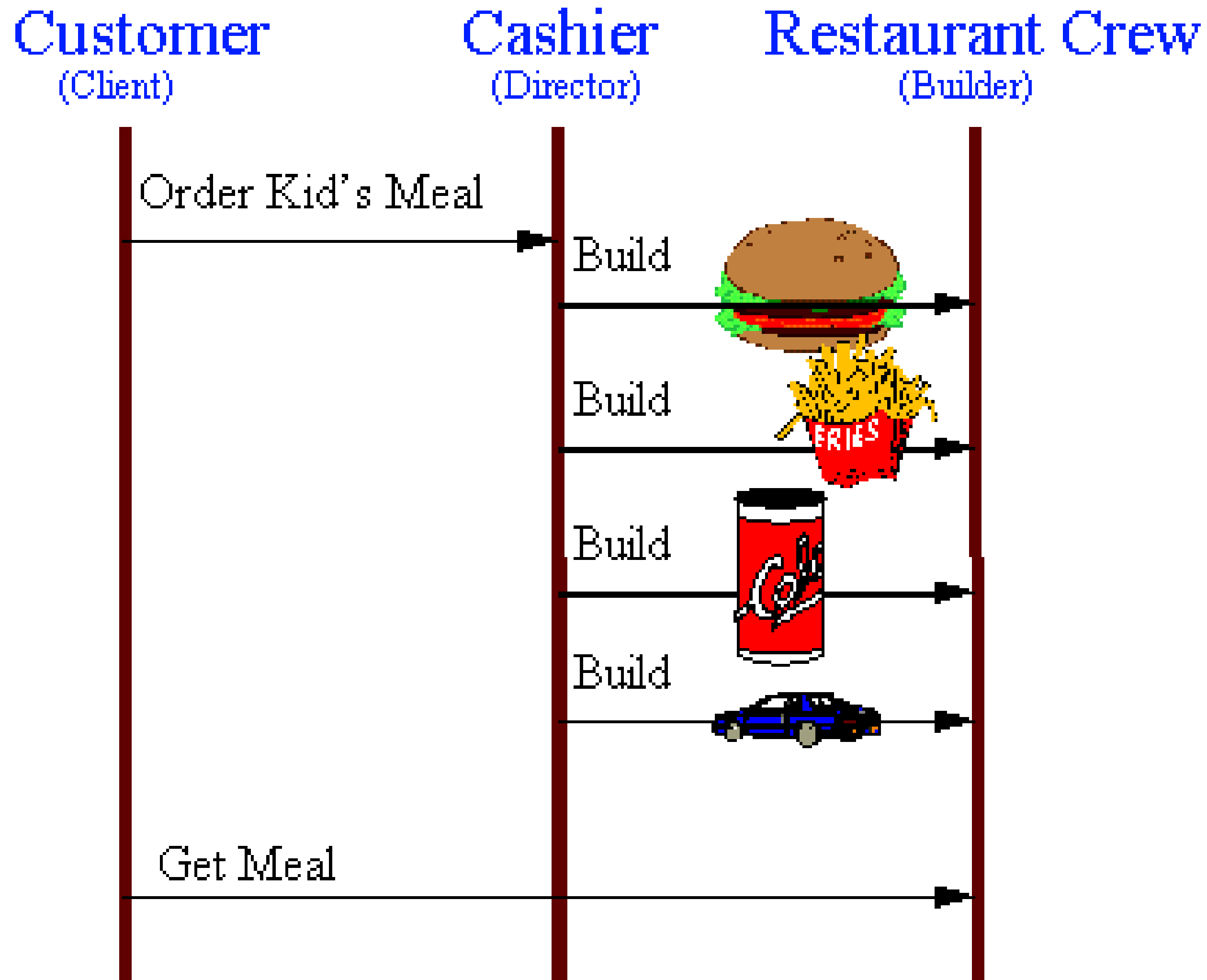
Builder: applicability

- Use the Builder pattern when
  - The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
  - The construction process must allow different representations for the object that's constructed

Builder: implementation

- The Builder interface
  - Must be general enough to allow construction of many products
- Abstract base class for all products?
  - Usually not feasible (products are highly different)
- Default implementation for methods of Builder?
  - "Yes": May decrease amount of code in ConcreteBuilders
  - "No:" May introduce silent bugs

Ex. found in the web: misleading, to be changed



GoF example: requirements

- A reader for the RTF (Rich Text Format) document exchange format should be able to convert RTF to many text formats.

- The reader might convert RTF documents into plain ASCII text or into a text widget that can be edited interactively.

- The problem:
  The number of possible conversions is
  open-ended. It should be easy to add a
  new conversion without modifying the reader.

GoF example: : Solution

- Configure the *RTFReader* class with a *TextConverter* object that converts RTF to another textual representation.
  - The *RTFReader* parses the RTF document,
  - When it recognizes an RTF token *t*

    → calls a *TextConverter* on *t*.

- *TextConverter* responsibilities:
  - perform data conversion.
  - represent the token in a particular format.
  - → Create and assemble a complex object.
  - → Hide this process.

- Subclasses of *TextConverter* specialize in different conversions and formats.

GoF example: Solution

RTFReader

ParseRTF()

builder

```
while (t = get the next token) {
    switch t.Type {
    CHAR:
        builder->ConvertCharacter(t.Char)
    FONT:
        builder->ConvertFontChange(t.Font)
    PARA:
        builder->ConvertParagraph()
    }
}
```

builders

TextConverter

ConvertCharacter(char)
ConvertFontChange(Font)
ConvertParagraph()

ASCIIConverter

ConvertCharacter(char)
GetASCIIText()

TeXConverter

ConvertCharacter(char)
ConvertFontChange(Font)
ConvertParagraph()
GetTeXText()

TextWidgetConverter

ConvertCharacter(char)
ConvertFontChange(Font)
ConvertParagraph()
GetTextWidget()

ASCIIText

TeXText

TextWidget

Ex: MazeBuilder (the abstract builder)

```
public class MazeBuilder {
   public void buildMaze(){ };
   public void buildRoom(int r){ };
   public void buildDoor(int d);{ }
   public Maze getMaze();{ }
}
```

This interface permits to create three things:

(1) the maze,

(2) rooms with a particular room number, (When a Room is built, also the walls are.)

(3) doors between numbered rooms.

The GetMaze operation returns the maze to the client.

Ex: MazeBuilder (the builders)

Subclasses of MazeBuilder will override these operations to return the maze that they build.

Ex.

public class StandardMazeBuilder implements MazeBuilder{ }

public class EnchantedMazeBuilder implements MazeBuilder{ }

Ex: MazeGame (director)

```
public class MazeGame {
   public Maze createMaze(MazeBuilder builder) {
            builder.buildRoom(1);  //also builds room's walls
            builder.buildRoom(2);
            builder.buildDoor(1,2);  // puts walls in common
            return builder.getMaze(); }

   public Maze createBigMaze(MazeBuilder builder) {
            builder.buildRoom(1);  … builder.buildRoom(100);
            builder.buildDoor(1,2);   … builder.buildDoor(99,100);
            return builder.getMaze(); }
}
```

Ex: MazeClient

```
public class Client {
   public static void main(String[] args) {
          Maze maze;
          MazeGame game = new MazeGame();  \\director
          MazeBuilder builder = new StandardMazeGame();
          game.createMaze(builder);
          maze = builder.GetMaze();
   }
}
```

Compare this version of CreateMaze with the original.

- Notice how the builder hides the internal representation of the Maze that is, the classes that define rooms, doors, and walls and how these parts are assembled to complete the final maze.
- Someone might guess that there are classes for representing rooms and doors, but there is no hint of one for walls.
- This makes it easier to change the way a maze is represented, since none of the clients of MazeBuilder has to be changed.

What is the difference between Builder Design pattern and Factory Design pattern?

- The Factory pattern can almost be seen as a simplified version of the Builder pattern.
- In the **Factory** pattern, the factory is in charge of creating various subtypes of an object depending on the needs.
- The user of a factory method doesn't need to know the exact subtype of that object. An example of a factory method createCar might return a Ford or a Honda typed object.
- In the **Builder** pattern, different subtypes are also created by a builder method, but the composition of the objects might differ within the same subclass.
- To continue the car example you might have a createCar builder method which creates a Honda-typed object with a 4 cylinder engine, or a Honda-typed object with 6 cylinders. The builder pattern allows for this finer granularity.

What is the difference between Builder Design pattern and Factory Design pattern? Cont'd

- Builder focuses on constructing a complex object step by step.
- Abstract Factory emphasizes a family of product objects (either simple or complex).
- Builder assembles and returns the product as a final step.
- Builder often builds a Composite.
- Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed.
- Sometimes creational patterns are complementary: Builder can use one of the other patterns to implement which components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementations.

Hemework (after having seen CoR)

- Use CoR to write a program that, given a number n, is able to return:
  1. the number of primes smaller than n
  2. the decomposition in prime factors of n
- Hint: build a chain of prime numbers, and assume n is smaller than 50.

- Use Builder to build the chain, in a situation where there are two kinds of handlers, and hence two chains (the client decides which chain to build):
  - one only dealing with prime factorization,
  - the other only counting the number of primes smaller than n