Creational Patterns

Builder

Motivation

- Create an object that requires too many optional fields
  1. Have multiple constructors: with 0,1,2,3,4,.. parameters
     - calling logic becomes more complex
     - Order of parameters difficult to remember by client programmer

```
Pizza(int size) { ... }
Pizza(int size, boolean cheese) { ... }
Pizza(int size, boolean cheese, boolean pepperoni) { ... }
Pizza(int size, boolean cheese, boolean pepperoni, boolean olive
... }
/*Client code*/ Pizza myPizza = new Pizza(12, true, false, true);
```

Motivation

- Create an object that requires too many optional fields
    1. Have multiple constructors: with 0,1,2,3,4,.. parameters

```
Pizza(int size) { ... }
Pizza(int size, boolean cheese) { ... }
Pizza(int size, boolean cheese, boolean pepperoni) { ... }
// Client Code is hard to read:
 Pizza myPizza = new Pizza(12, true, false, true);
                                          // What do these booleans mean?
```

- cluttered class and a hard to read client code

Motivation Example

- Create an object that requires too many optional fields

1. Have multiple constructors: with 0,1,2,3,4,.. Parameters
    - calling logic becomes more complex
2. Write an enormous constructor with a lot of functional logic.
    - too many ifs to check existence of valid parameter values
    - the code becomes more complex and harder to debug
    - Problems subclassing due to selection logic

Motivation Example

- Create an object that requires too many optional fields

1. Have multiple constructors: with 0,1,2,3,4,.. Parameters
2. Write an enormous constructor with a lot of selection logic.
3. Have a null constructor and setter methods
   - Better than before but….

Motivation Example

3. Have a null constructor and setter methods

```
public class Pizza {
public Pizza() { } // Default constructor
public void setSize(int size) { ... }
public void setCheese(boolean hasCheese) { ... }
public void setPineapple(boolean hasPineapple) { ... }
….
// Client Code:
Pizza myPizza = new Pizza(); myPizza.setSize(12);
myPizza.setCheese(true);
 // What if the pizza is passed to another method here? It's incomplete!
myPizza.setOlive(true); //now complete
```

- The object is in an inconsistent state during its construction
- since the class has setters, we can't create immutable objects

Motivation Example

- Create an object that requires too many optional fields

1. Have multiple constructors: with 0,1,2,3,4,.. Parameters
2. Write an enormous constructor with a lot of selection logic.
3. Have a null constructor and setter methods
   - Better than before but….
   - Object in an unstable state may be used causing errors
     - Assume 5th value is required. Before the 5th set, the object is in unstable state. Some part of client might see the objects in 4th state and assume it is done.
   - Need extra effort in concurrency to ensure thread safety
   - What if order of set methods is important?
   - Cannot create immutable objects

How about..

- Construct the object step by step
- But hide the object during the creation and reveal only after the object is in a stable state
- i.e. encapsulate the creation

- Motivation1: staged object creation

Complex objects

- Complex objects are made of parts made of other objects that need special care when being built.
- An application might need a mechanism for building complex objects that is independent from the ones that make up the object
- Motivation2 : complex object
- The same parts, in a different assembly may result in different complex object

Builder

- **Intent:** Separate the **construction** of a complex object from its **representation** so that the same construction process can create different representations.
  - Build different complex objects from the same set of component parts

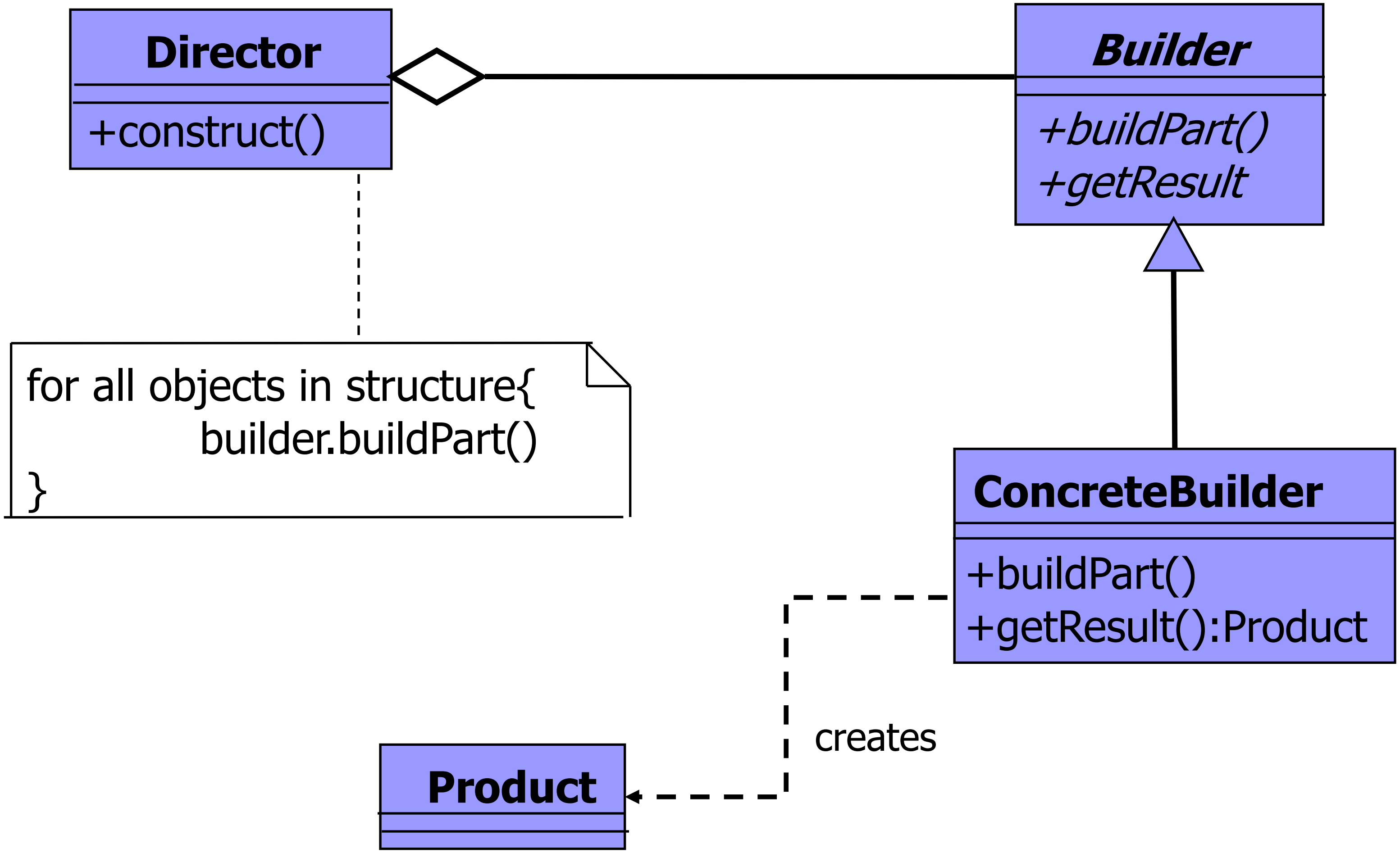
Similar to Bridge, but Bridge is not about creating objects

Does this sound familiar?

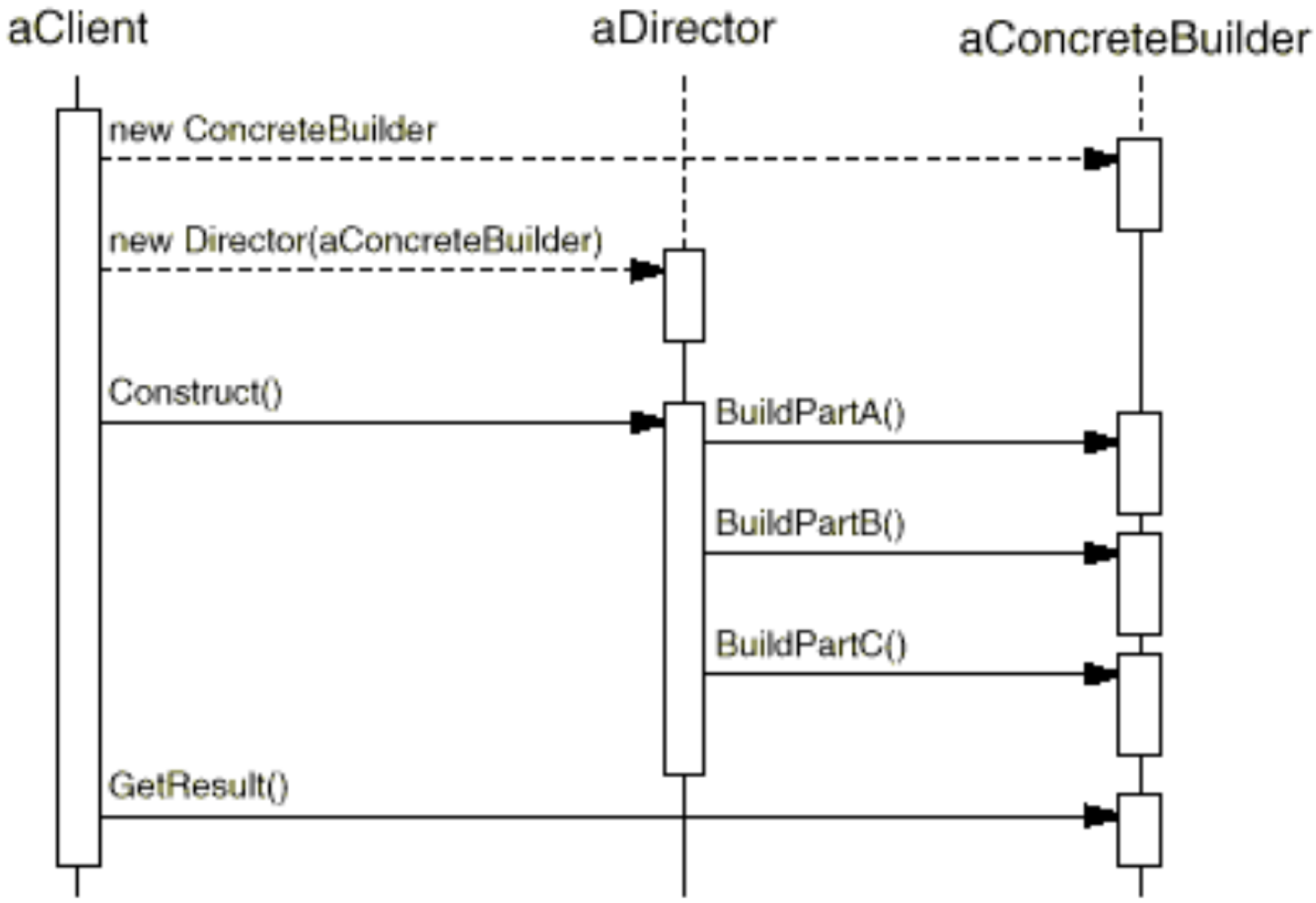
Builder

- **Applicability**
  - The algorithm for creation is independent of the parts and how they are assembled
  - Different construction processes results in different representation
  - building a complex object from possibly multiple different sources

Builder - Structure



Collaborations



Back to Pizza Example

```
class Recipe{ // Director
 public:
    Recipe(PizzaBuilder* b);
    void
setBuilder(PizzaBuilder*b);
    virtual void make(int s) {
        builder->buildPizza(s);
        builder->addCheese();
     }
  private: PizzaBuilder* builder;
}
```

```
class PizzaBuilder{
 public:
    PizzaBuilder();
    virtual void buildPizza(int
size);
    virtual void addCheese();
    virtual void addOlives();
    //…
    virtual Pizza* getResult();
  private: Pizza* pizza;
}
```

~~Pizza(int size) { ... }~~
~~Pizza(int size, boolean cheese) { ... }~~
~~Pizza(int size, boolean cheese, boolean pepperoni) { ... }~~
~~Pizza(int size, boolean cheese, boolean pepperoni, boolean olives) { ...~~
~~}~~

Back to Pizza Example

```
class Recipe{
 public:
    Recipe(PizzaBuilder* b);
    void
setBuilder(PizzaBuilder*b);
    virtual void make(int s) {
        builder->buildPizza(s);
        builder->addCheese();
     }
  private: PizzaBuilder* builder;
}
```

```
class PizzaBuilder{
 public:
    PizzaBuilder();
    virtual void buildPizza(int
size);
    virtual void addCheese();
    virtual void addOlives();
    //…
    virtual Pizza* getResult();
  private: Pizza* pizza;
}
```

```
//client code
PizzaBuilder builder=new PizzaBuilder();
Recipe r (builder);        r.make(12);
Pizza* cheesepizza= builder-
>getResult();
```

Exercise: Bike

- Build a bike
  - Make wheels, steering, etc..
- Participants
  - Client: asks for a bike
  - Director: instructs how to build the bike
  - Builder: builds the parts

Building a Bike

```
class Client{…
 void mymethod(Builder* builder){
  contractor.construct(builder);
  Bike* mybike= builder-
   >getResult();
 }
}
class Director{
 public:
 construct(Builder* builder){
     builder->makeBike();
     builder->buildWheel(1);
     builder->buildWheel(2);
     builder->makeSteering();
 }
    }
```

```
class Builder{
    virtual void makeBike()=0;
    virtual void buildWheel(int)=0;
    virtual void makeSteering()=0;
    virtual void trainingWheels() =0;
    virtual Bike* getResult()=0;
}
```

- Builder hides the internal representation of the product
- You guess that there are classes for wheels and steering
- But no hint whether there are classes for chains or gears.

Note: destructor, copy constructor etc not shown for brevity

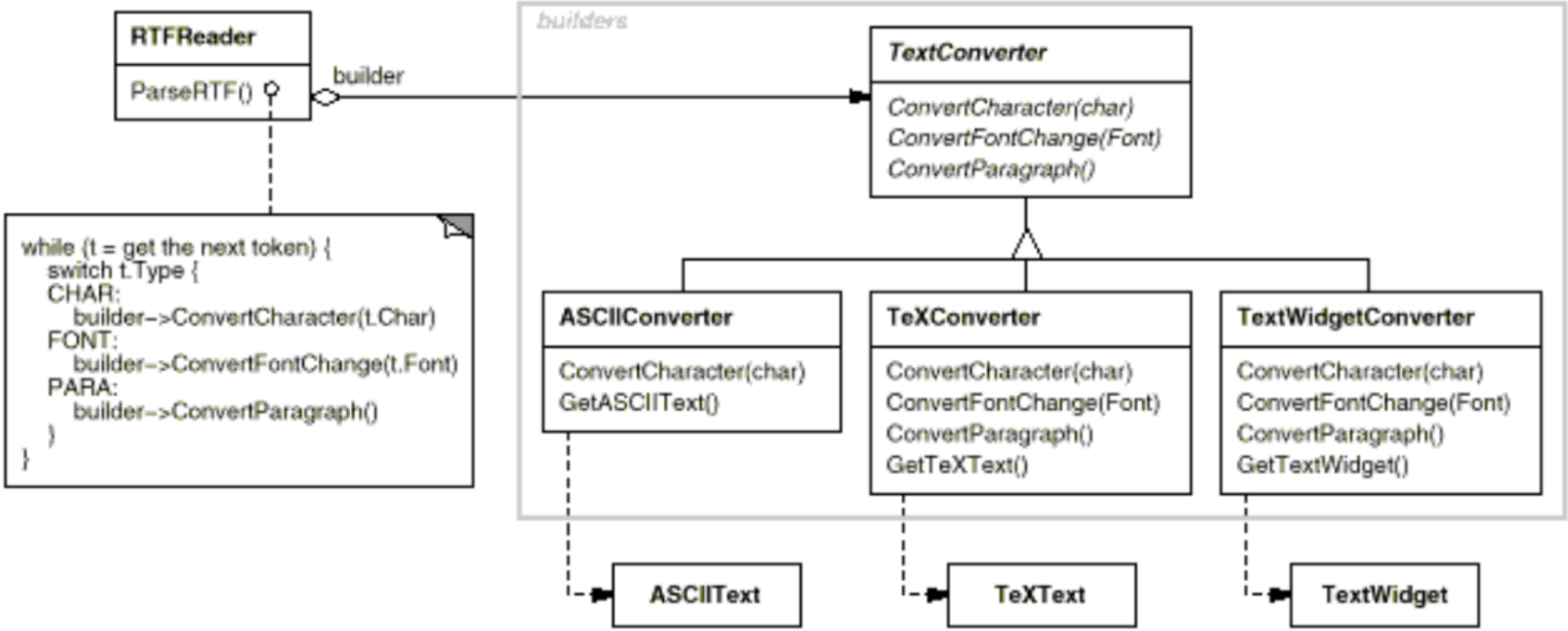
Building a Bike

```
class DefaultBuilder: public Builder{
  private: Bike* bike;
  public:
    void makeBike() {bike=new Bike();}
     //alt:prototype
    void makeSteering (){…};
    void makeHelperWheels(){…};
    void buildWheel(int r){
        if(bike->hasWheel(r)) return;
        Wheel* w=new Wheel(r);
        w->setGear(new Gear(4));
        bike->add(w);
    };

    Bike* getResult(){return bike; }
}
```

- You may guess that there are classes for wheels etc
- But no hint whether there are classes for gears.

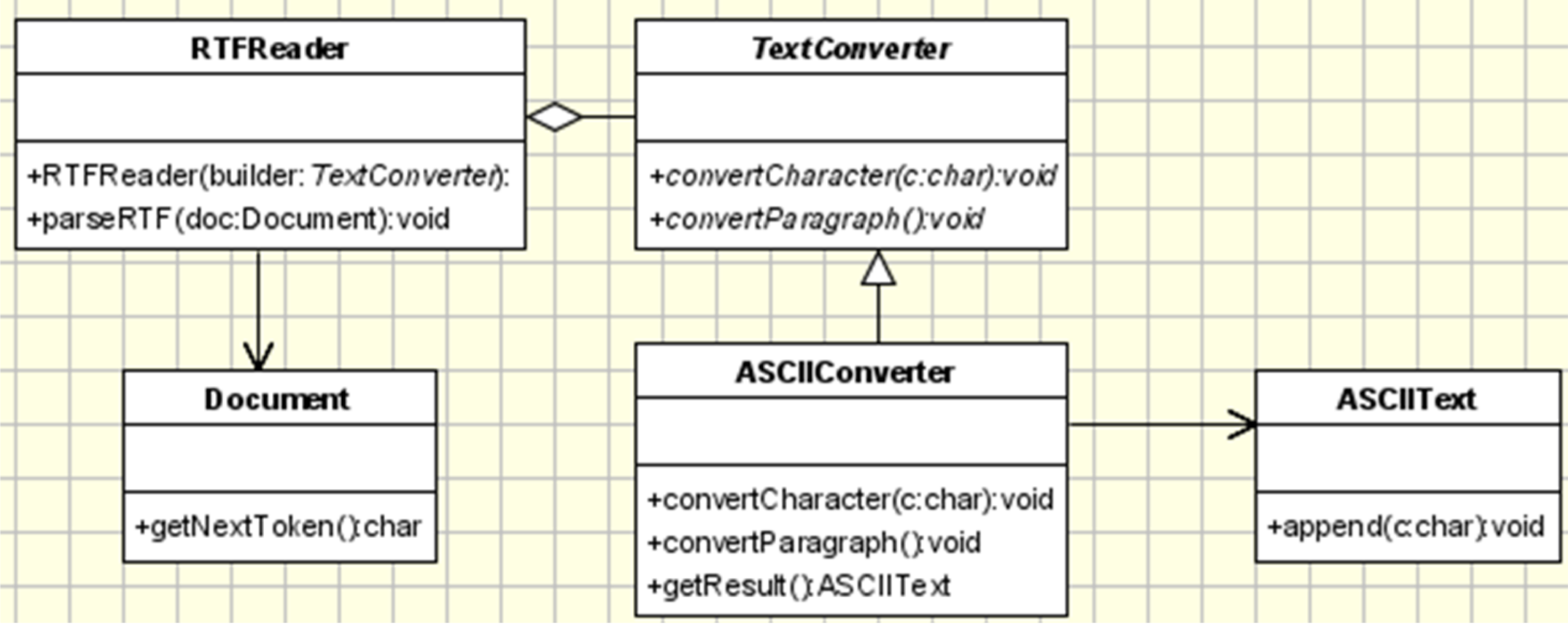
GoF Doc format converter



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

Example 3: Doc format converter

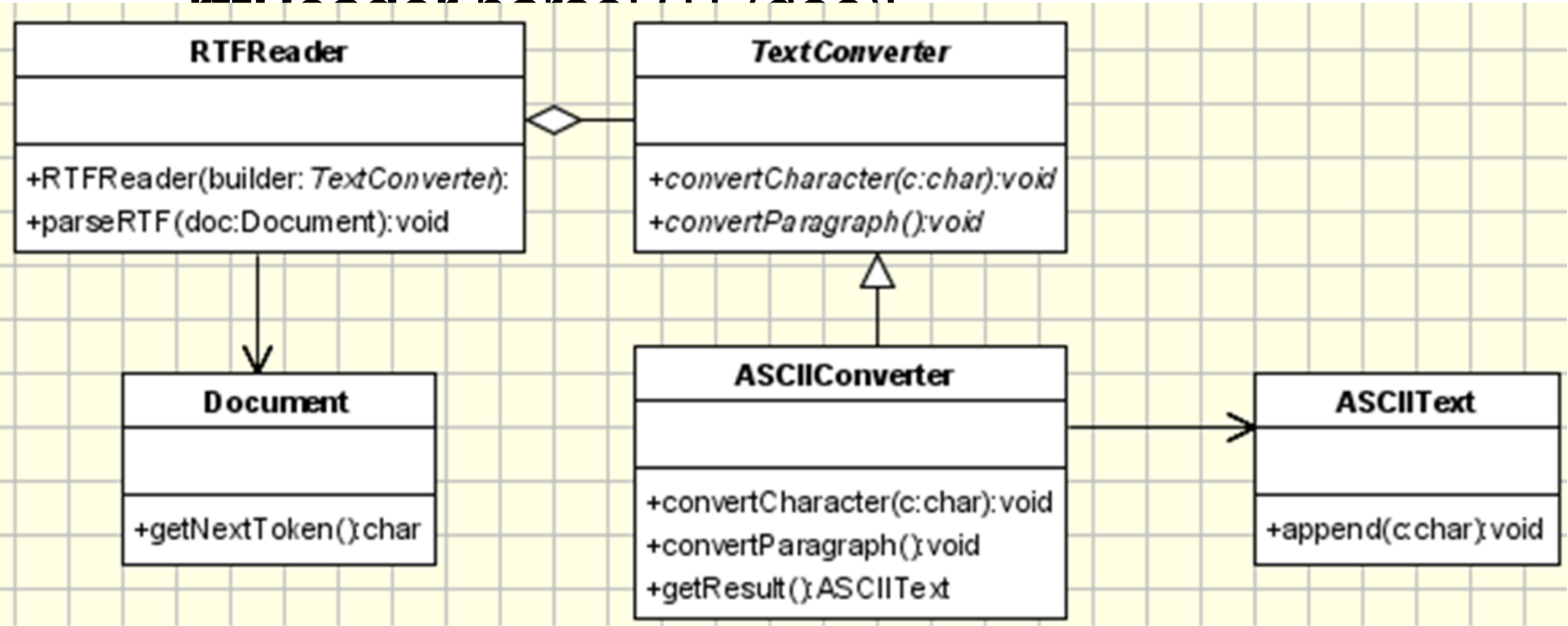
- Get an RTF document, transform it



https://www.oodesign.com/builder-pattern

Example 3: Client code use

```
void createASCIIText(Document doc){
        ASCIIConverter asciiBuilder = new
ASCIIConverter();
        RTFReader rtfReader = new
RTFReader(asciiBuilder);
        rtfReader.parseRTF(doc);
```



```
//Abstract Builder
public interface TextConverter{
  public void convertCharacter(char c);
  public void convertParagraph();
}
```

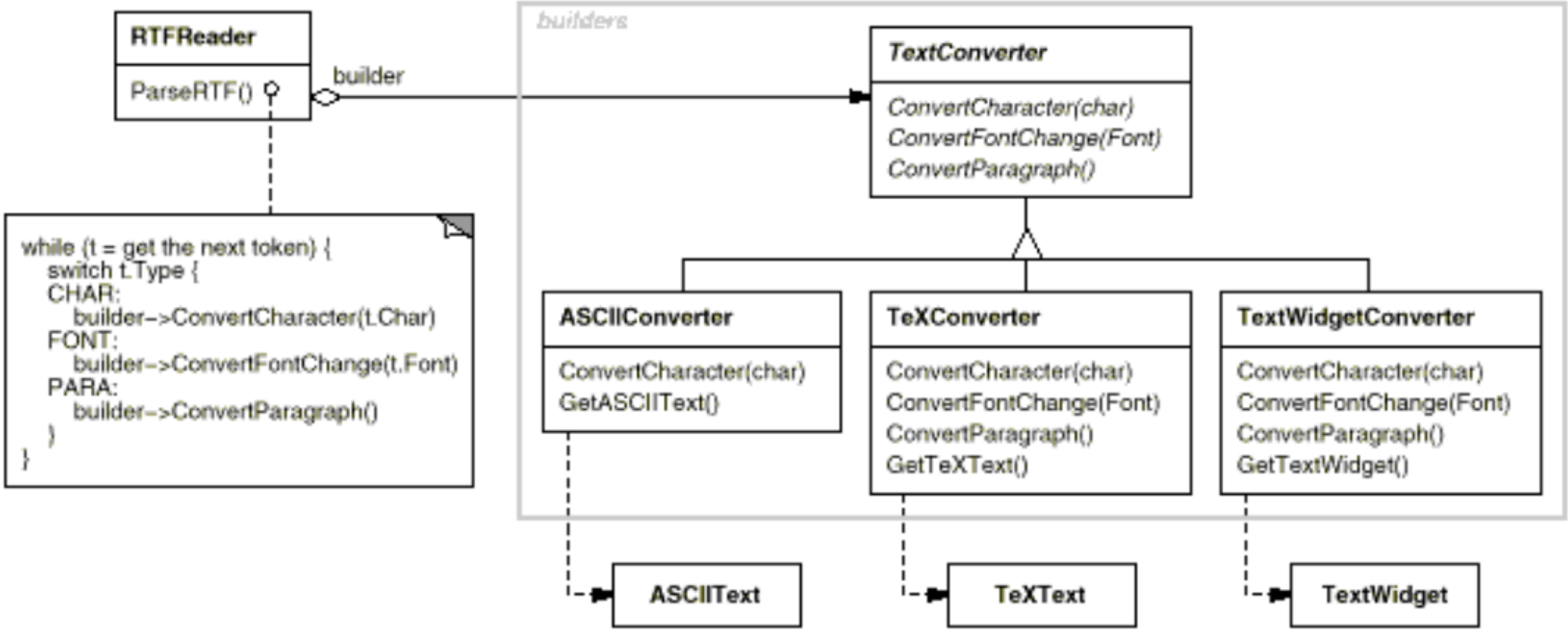
```
//Director
class RTFReader{
    private static final char EOF='0’;
    final char CHAR='c';  final char PARA='p';  char t;
    private TextConverter builder;
    public RTFReader(TextConverter obj){   builder=obj; }
    public void parseRTF(Document doc){
            while ((t=doc.getNextToken())!= EOF){
                  switch (t){
             case CHAR: builder.convertCharacter(t); break;
             case PARA: builder.convertParagraph(); break;
            }
     }
}
```

```
//Concrete Builder
public class ASCIIConverter implements TextConverter{
     private ASCIIText asciiTextObj;//resulting product

    /*converts a character to target representation
     and appends to the resulting*/
     public  void convertCharacter(char c){
             char asciiChar = new Character(c).charValue(); //get char value
             asciiTextObj.append(asciiChar);
             }
    public void convertParagraph(){/*nothing to do*/}
    //These details are hidden from the Director.

    public ASCIIText getResult(){  return asciiTextObj;       }
}
```

GoF Doc format converter



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

Use builder when a class

- Has complex internal structure
  - especially one with a variable set of related objects
- Has attributes that depend on each other.
  - E.g. while building an order, set the country before billing since it may change the pricing
  - Builder can enforce **staged construction** of a complex object.
  - This would be required when the Product attributes depend on one another.

- The Builder coordinates the assembly of the product object:
  - creating resources,
  - storing intermediate results,
  - and providing functional structure for the creation.
- Additionally, the Builder can acquire system resources required for construction of the product object.
- Example: Business objects
  - frequently require data from a database for initialization
  - might need to associate with several other business objects to accurately represent the business model as soon as it’s created.

Immutable Complex Objects

- **Immutable**: once created the fields cannot be reassigned
    - No setters
- Builder helps creating immutable complex objects
- Mechanism: Have a static builder class as an inner class of the Product.
    - Caution: not as flexible as the presented version of builder

```
class Entity{
  private final int f1, f2;
  public static class Builder{
     private int requiredField;
     private int optionalField;
     public Builder (int required){
           this.requiredField=required;}
    public Builder option1(int
optionalVal){
           optionalField=optionalVal;
           return this;
     }
     public Entity build(){
       return new Entity(this);}
  }
 private Entity(Builder builder){
       f1=builder.requiredField;
       f2=builder.optionalField;   }
```

- Usage

```
Entity p1=new
          Entity.Builder(12)
          .option1(3)
          .build();
```

No one can set/change the fields of this object

- Did anyone used the StringBuilder?

C++

```
class Entity {
  public:
     class Builder {
         private:  int requiredField,
optionalField;
       public:
            Builder(int required) :
                   requiredField(required) {}
            Builder& option1(int optionalVal)
{
              optionalField = optionalVal;
              return *this; }
            Entity build() { return Entity(*this);
}
       };
  private: const int f1; const int f2;
  Entity(const Builder& builder) :
             f1(builder.requiredField),
             f2(builder.optionalField) {}
```

- Usage

```
Entity p1=
          Entity::Builder(12)
          .option1(3)
          .build();
```

No one can set/change the fields of this object

Side note: testing

- Have you seen such code in testing?

```
m.expects(once())
    .method("someMethod")
    .with(eq(1), eq(2))
    .returns("someResponse");
```

This is jmock recording a scenario to be executed while testing a function.

Builder in action!

Build method chaining

- Example:

```
m.expects(once())
   .method(“someMethod")
   .with(eq(1), eq(2))
   .returns("someResponse");
```

- Currently, most Builder implementations make their methods to return the builder object itself for method chaining
  - public StringBuilder append(String); //in StringBuilder

Builder Consequences -1

- Enables to vary a product’s internal representation
  - Change the Builder, get a Product with different representation
- Isolates code for construction and representation
  - Clients do not know component classes that does not appear in the interface
  - Different directors can reuse Builder to create Product variants from the same parts
- Fine control over the construction process
  - Construct step by step and retrieve the product Only when it is finished
    - Director has fine control over which steps to execute

Consequence -2

- For objects that require phased creation the Builder acts as a higher-level object to oversee the process. It can coordinate and validate the creation of all resources and if necessary, provide a fallback strategy if errors occur.
- For objects that need existing system resources during creation, such as database connections or existing business objects, the Builder provides a central point to manage these resources.
  - a single point of creational control for its product,
  - easier for clients: since they need only access the Builder object to produce a resource.

Consequence -3

- Drawback: tight coupling among the Builder, its product, and any other creational delegates used
  - Changes that occur for the product created by the Builder often result in modifications for both the Builder and its delegates.

Implementation issues

- Which methods to be in the Builder class?
    - general enough to allow the construction of products for all kinds of concrete builders.
- Assembly: Simple append mostly
    - Construction process may require previous parts
        - E.g. buildnode() returns the node created so that process can ask buildParent(node)
- Chaining build methods via returning the Builder
- Builder interface or class with empty methods
    - Have empty methods, subclass overrides only the supported build methods
- No Product interface: Not very similar, no need

Known uses

- java.lang.StringBuilder
    - append() and toString() methods
- java.lang.StringBuffer
- java.util.stream.Stream.Builder

Builder vs Abstract Factory

- Builder construct objects step by step
  - instruct the builder how to create the object and then asks for the result.
  - How the class is put together is up to the Builder
  - Returns the product as final step
- Abstract Factory returns the object in 1 shot
  - is focused on family of product objects
  - Product is returned immediately

Related Patterns

- Create Composite pattern objects with builder

Builder

- **Intent:** Separate the **construction** of a complex object from its **representation** so that the same construction process can create different representations.

Creational Pattern Comparison

- Abstract Factory
  - Emphasis on families of product objects
- Builder
  - A director instructs step by step construction
- Prototype
  - Cloning and filling in the details of the cloned class to behave as desired
    - you need to do cloning of the same object and may want try out different operation
- Factory method
  - moves the object creation required by a class to its subclass
- Singleton
  - Only one object with global access