

CSCI 253

Object Oriented Design:
Builder Pattern

George Blankenship

Builder Pattern

George Blankenship

1

Overview

Creational Patterns

- Singleton
- Abstract factory
- Factory Method
- Prototype
- Builder

Structural Patterns

- Composite
- Facade
- Proxy
- Flyweight
- Adapter
- Bridge
- Decorator

Behavioral Patterns

- Chain of Respons.
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Builder Pattern

George Blankenship

2

The Elements of a Design Pattern

- A pattern name
- The problem that the pattern solves
 - Including conditions for the pattern to be applicable
- The solution to the problem brought by the pattern
 - The elements (classes-objects) involved, their roles, responsibilities, relationships and collaborations
 - Not a particular concrete design or implementation
- The consequences of applying the pattern
 - Time and space trade off
 - Language and implementation issues
 - Effects on flexibility, extensibility, portability

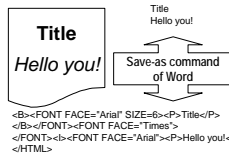
Builder Pattern

George Blankenship

3

The Builder Pattern: The Problem

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



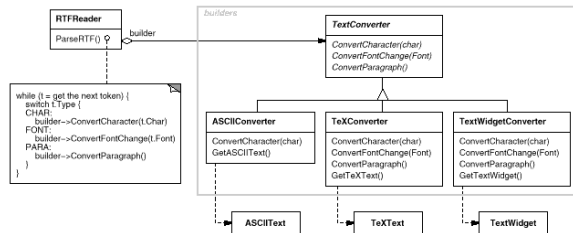
- a RTF reader that can convert into many different formats
- a parser that produces a complex parse tree

Builder Pattern

George Blankenship

4

RTF Reader



Builder Pattern

George Blankenship

5

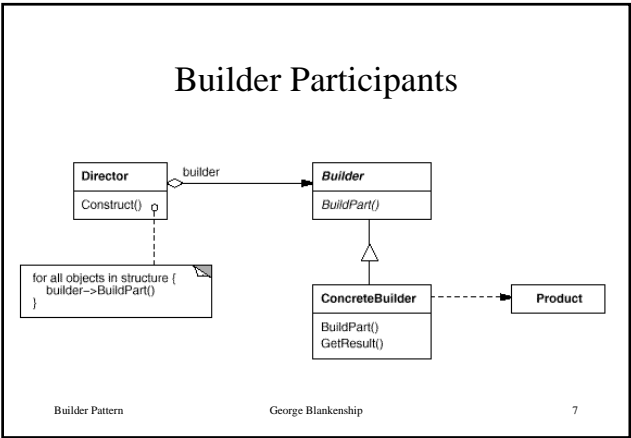
The Builder Pattern Participants

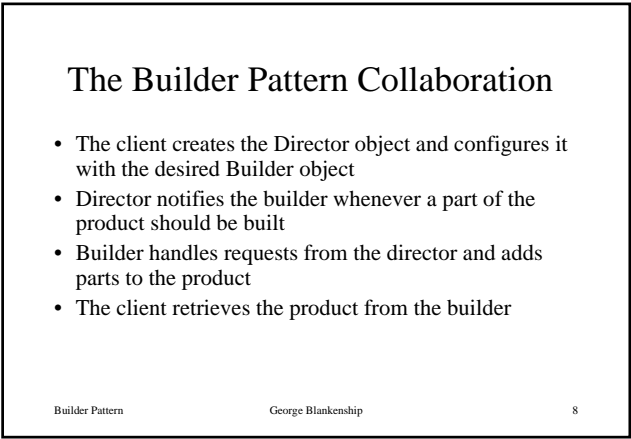
- **Builder:** specifies an abstract interface for creating parts of a Product
- **ConcreteBuilder:**
 - constructs and assembles parts of the Product by implementing the Builder interface
 - defines and keeps track of the representation it creates
 - provides an interface for retrieving the product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

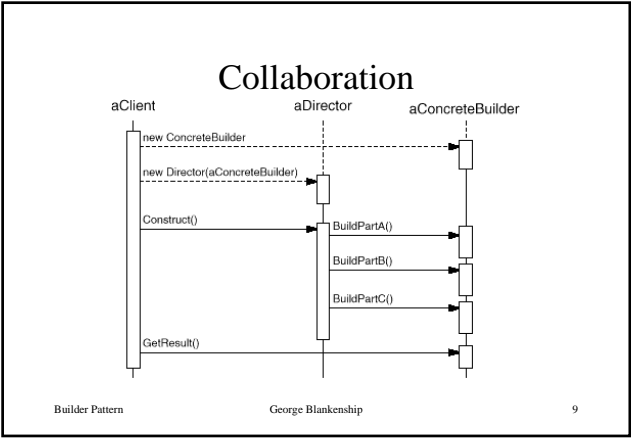
Builder Pattern

George Blankenship

6

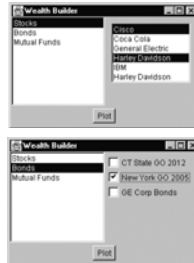






Multichoice GUI

- We would like to have a display that is easy to use for either a large number of funds (such as stocks) or a small number of funds (such as mutual funds).
- We want some sort of a multiple-choice display so that we can select one or more funds to plot.
- If there is a large number of funds, we'll use a multi-choice list box and if there are 3 or fewer funds, we'll use a set of check boxes.
- We want our Builder class to generate an interface that depends on the number of items to be displayed, and yet have the same methods for returning the results.



Builder Pattern

George Blankenship

10

multiChoice Class

```
abstract class multiChoice {
//This is the abstract base class that are the parent for the listbox and checkbox
choice panels
Vector choices; //array of labels
//-----
public multiChoice(Vector choiceList) {
    choices = choiceList; //save list
}
//to be implemented in derived classes
abstract public Panel getUI(); //return a Panel of components
abstract public String[] getSelected(); //get list of items
abstract public void clearAll(); //clear selections
}
```

Builder Pattern

George Blankenship

11

Choice Panel Classes

- ```
class listBoxChoice extends multiChoice
 - Create a list box for a large number of choices
class checkBoxChoice extends multiChoice
 - Create a set of check boxes for small number of
 choices
```

Builder Pattern

George Blankenship

12

## Panel Generation

```
class choiceFactory {
 multiChoice ui;
 //This class returns a Panel containing a set of choices displayed by one of several UI
 methods.
 public multiChoice getChoiceUI(Vector choices) {
 if(choices.size() <=3) //return a panel of checkboxes
 ui = new checkBoxChoice(choices);
 else //return a multi-select list box panel
 ui = new listBoxChoice(choices);
 return ui;
 }
}
```

Builder Pattern

George Blankenship

13

---

---

---

---

---

---

---

---

## The Builder Pattern Consequences

- + Lets you vary the product's internal representation: the directors uses the abstract interface provided by the builder for constructing the product; to change the products representation, just make a new type of builder
- + Allows reuse of the ConcreteBuilders: all code for construction and representation is encapsulated; different directors can use the same ConcreteBuilders
- + Gives finer control over the construction process: in other creational patterns, construction is often in one shot; here the product is constructed step by step under the director's guidance giving fine control over the internal structure of the resulting product

Builder Pattern

George Blankenship

14

---

---

---

---

---

---

---

---

## The Builder Pattern Implementation

- Assembly and construction interfaces:
  - The Builder interface must be general enough to allow the construction of products for all kinds of ConcreteBuilders
  - The model for construction and assembly is a key design issue
- Why no abstract class for products?:
  - In the common case, the products can differ so greatly in their representation that little is to gain from giving different products a common parent class
  - Because the client configures the Director with the appropriate ConcreteBuilder, the client knows the resulting products
- Empty methods as default in Builder:
  - In C++ the build methods are intentionally not pure virtual member functions but empty methods instead; this allows clients to overwrite only the operations they are interested in

Builder Pattern

George Blankenship

15

---

---

---

---

---

---

---

---