CptS 487
Software Design and Architecture

Lesson 13

Design Patterns 4:

Builder & Singleton

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

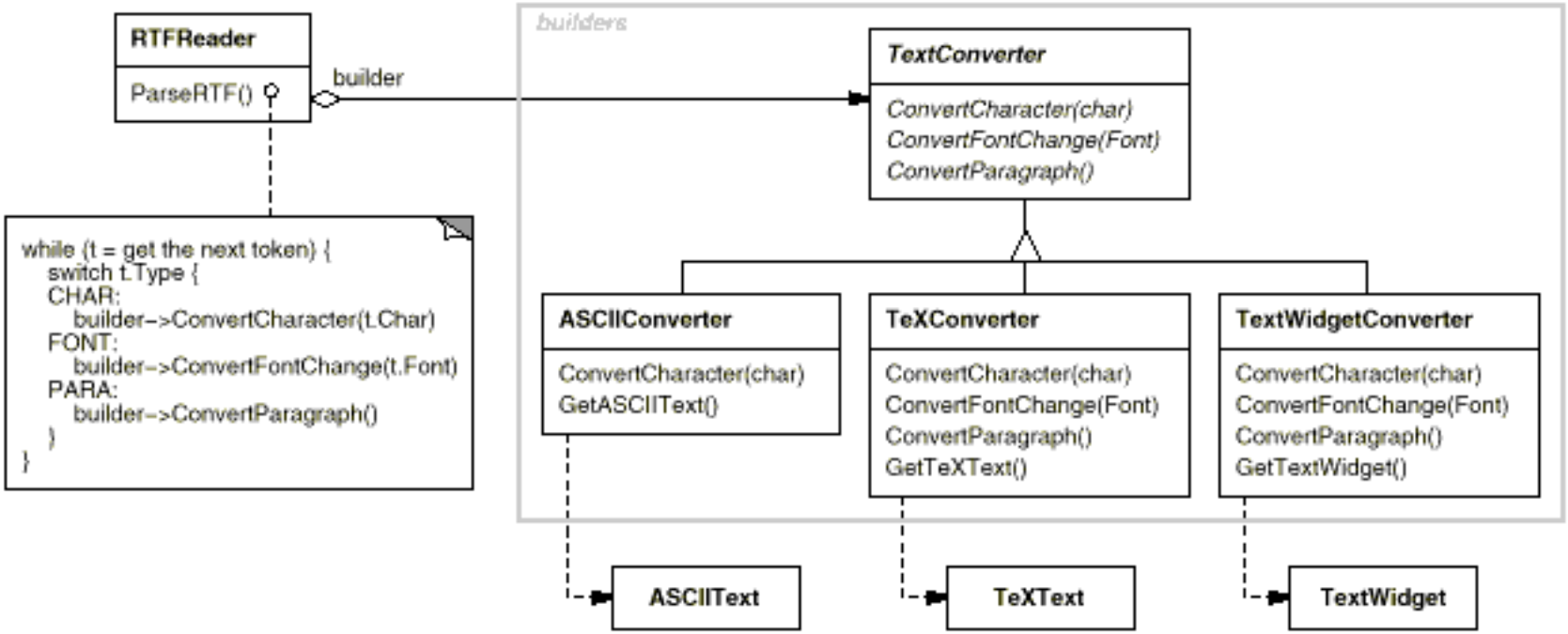
Instructor: Bolong Zeng

Overview

- Introducing two other creational design pattern
  - Builder Pattern
    - Understand the difference between Builder and Factory/Abstract Factory.
  - Singleton Pattern
    - Two ways of ensuring an object to be a singleton.
    - Typical applications.

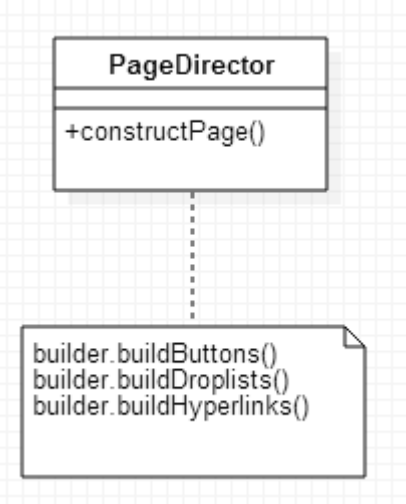
3. Builder (Object creational pattern)

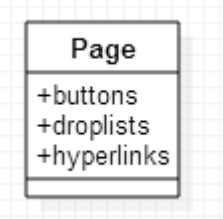
- Motivation



Example

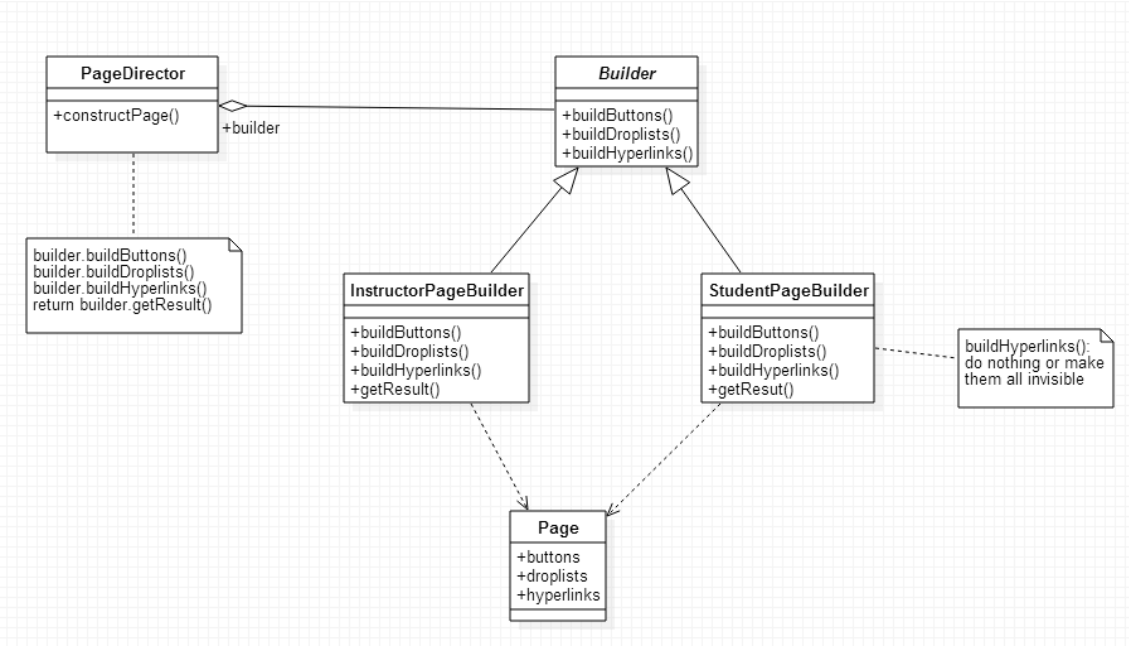
- Build different web-pages for different actors.
  - Blackboard: Student, Instructor
  - Enable and disable different contents with respect to the user's identity. For instance, only the instructor could see the hyperlinks





Example

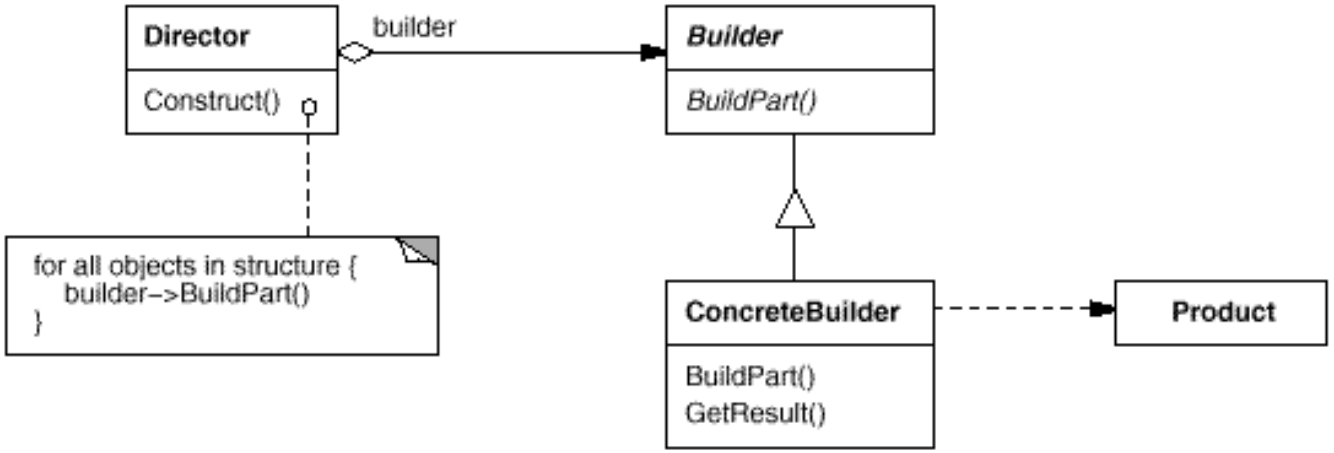
- Build different web-pages for different actors.
  - Blackboard: Student, Instructor
  - Enable and disable different contents with respect to the user's identity. For instance, only the instructor could see the hyperlinks



3. Builder (Object creational pattern)

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

- Structure

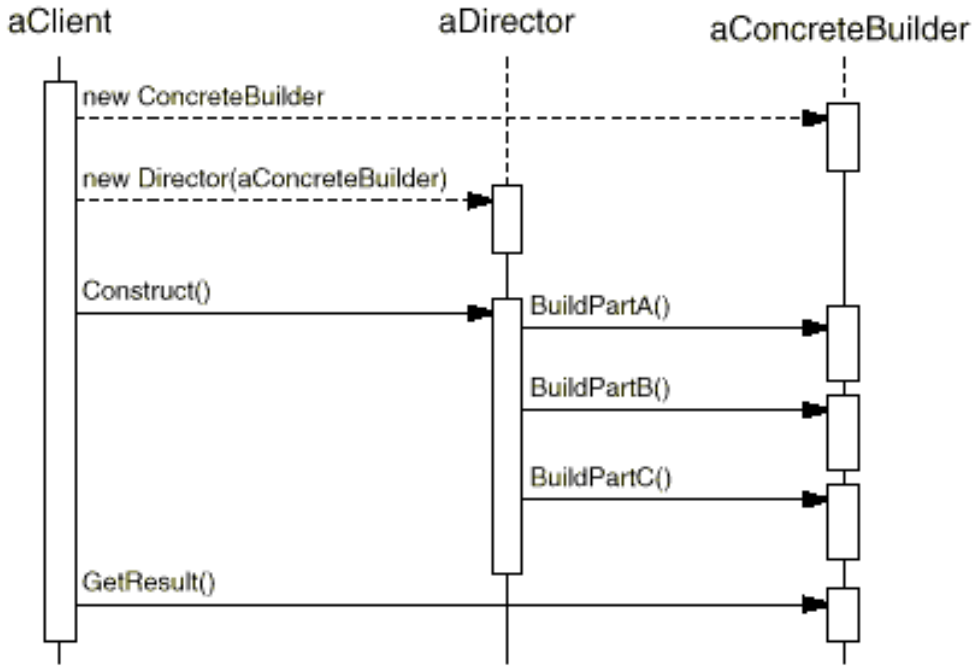


3. Builder (Object creational pattern)

- 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.

3. Builder (Object creational pattern)

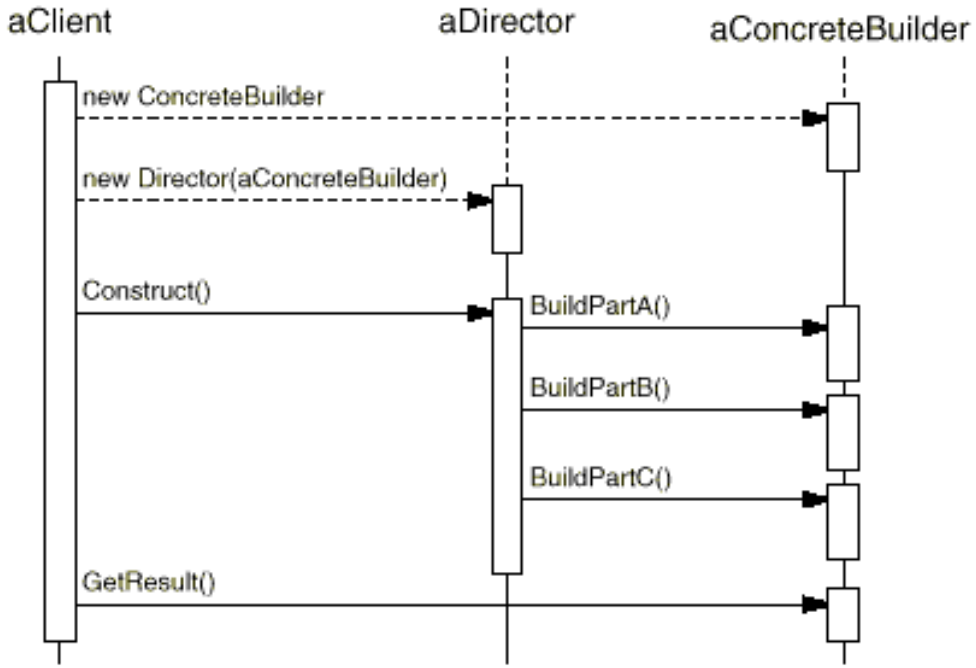
- Collaborations
  - The client creates the Director object and configures it with the desired Builder object.
  - Director notifies the builder whenever a part of the product should be built.
  - Builder handles requests from the director and adds parts to the product.
  - The client retrieves the product from the builder.



The interaction diagram illustrates how Builder and Director cooperate with a client.

Builder vs Factory Method

- A builder is useful when several steps are needed to build an object
- A factory is used when the factory can easily create the entire object within one method call
- Example: Building an XML document. Which pattern to use?



The interaction diagram illustrates how Builder and Director cooperate with a client.

Builder vs Abstract Factory

- *From oodesign.com:
- Abstract Factory
  - *In the case of the Abstract Factory, the client uses the factory's methods to create its own objects.
  - *Objects/Products are derived from a common type.
- Builder
  - *In the Builder's case, the Builder class is instructed on how to create the object and then it asked for it, but the way that the class is put together is up to the Builder class.
  - The parts don't have to be "similar".

- In reality, you often go through an "evolution" process:
  - Factory (prototype stage, early requirements/features)

  -> Abstract Factory (later development, have certain restrictions)

  -> Builder (complex and flexible product made up of many "parts")

Creational “vs.” Structural

- So far we have:
  - Creational:
    - Factory, Abstract Factory, Builder
  - Structural:
    - Composite
    - More to come.

- Are they mutually exclusive?

- In fact, they are often used to work together.
  - Builder typically is used to build composite-like class structures.
  - Revisit the “Maze” example earlier.
  - More online resources and textbooks.

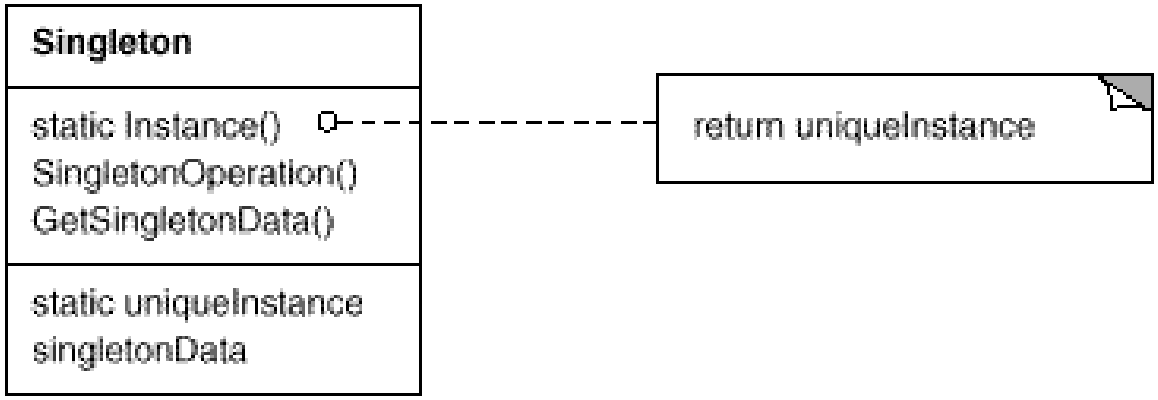
4. Singleton (Object creational pattern)

- Intent
  - Ensure a class only has one instance, and provide a global point of access to it

- Motivation
  - Sometimes we want just a single instance of a class to exist in the system
  - For example, we want just one window manager. Or just one factory for a family of products.
  - We need to have that one instance easily accessible
  - And we want to ensure that additional instances of the class can not be created

4. Singleton (Object creational pattern)

- Structure



- Consequences
  - Benefits
    - Controlled access to the sole instance
    - Can be extended to permits a variable number of instances

Singleton Implementation

- How do we implement the Singleton pattern?
  - We'll use a **static** method to allow clients to get a reference to the single instance and we'll use a **private constructor**!
  - Static method:
    - It is a method which belongs to the class and not to the object(instance)
    - A static method can be accessed directly by the class name and doesn't need any object

Singleton Implementation

— Class Singleton is an implementation of a class that only allows one instantiation.

```
public class Singleton {

   private static Singleton uniqueInstance = null;

   // An instance attribute.
   private int data = 0;

   public static Singleton instance() {
        if(uniqueInstance == null) uniqueInstance = new Singleton();
        return uniqueInstance;
   }

   private Singleton() {}

   // Accessors and mutators here!
}
```

The private reference to the one and only instance.

Returns a reference to the single instance. Creates the instance if it does not yet exist. (This is called lazy instantiation.)

The Singleton Constructor. Note that it is private! No client can instantiate a Singleton object!

Singleton Implementation

- Here's a test program:

```
public class TestSingleton {

  public static void main(String args[]) {

    Singleton s = Singleton.instance();

    s.setData(34);

    System.out.println("First reference: " + s);
    System.out.println("Singleton data value is:"+s.getData());


    s = null;
    s = Singleton.instance();

    System.out.println("\nSecond reference: " + s);
    System.out.println("Singleton data value is: " +
    s.getData());
  }
}
```

Get a reference to the single instance of Singleton

Set the data value

Get another reference to the Singleton. Is it the same object?

Singleton Implementation

- The test program output:

```
First reference: Singleton@1cc810
Singleton data value is: 34

Second reference: Singleton@1cc810
Singleton data value is: 34
```

4. Singleton (Object creational pattern)

- Note that the singleton instance is only created when needed.
- This is called lazy instantiation.
- What if two threads concurrently invoke the instance() method? Any problems?
  - Two instances of the singleton class could be created!
- How could we prevent this? Several methods:
  - Make the instance() synchronized. Synchronization is expensive, however, and is really only needed the first time the unique instance is created.
  - Do an eager instantiation of the instance rather than a lazy instantiation.

Singleton Implementation

- Here is Singleton with eager instantiation.
  - We'll create the singleton instance in a static initializer. This is guaranteed to be thread safe.

```
/*Class Singleton is an implementation of a class that only allows one
instantiation*/
public class Singleton {

  private static Singleton uniqueInstance = new Singleton();

  // An instance attribute.
  private int data = 0;

  public static Singleton instance() {
       return uniqueInstance;
  }

  private Singleton() {}

  // Accessors and mutators here!
}
```

The private reference to the one and only instance. Let's eagerly instantiate it here

Returns a reference to the single instance

The Singleton Constructor. Note that it is private! No client can instantiate a Singleton object!

Singleton With Subclassing

- What if we want to be able to subclass Singleton and have the single instance be a subclass instance?
  - For example, suppose MazeFactory had subclasses EnchantedMazeFactory and AgentMazeFactory. We want to instantiate just one factory, either an EnchantedMazeFactory or an AgentMazeFactory.

- How could we do this? Several methods:
  - Have the static instance() method of MazeFactory determine the particular subclass instance to instantiate. This could be done via an argument or environment variable. The constructors of the subclasses can not be private in this case, and thus clients could instantiate other instances of the subclasses.

  or

  - Have each subclass provide a static instance() method. Now the subclass constructors can be private.

Singleton With Subclassing Method 1

- Method 1: Have the MazeFactory instance() method determine the subclass to instantiate

```
/* Class MazeFactory is an implementation of a class that  only allows
one instantiation of a subclass. */

public abstract class MazeFactory {

   private static MazeFactory uniqueInstance = null;


   protected MazeFactory() {}
```

The private reference to the one and only instance.

The MazeFactory constructor. If you have a default constructor, it can not be private here!

continued on the next slide...

Singleton With Subclassing Method 1 (Continued)

Return a reference to the single instance.
If instance not yet created, create "enchanted"
as default.

```
  public static MazeFactory instance() {
    if (uniqueInstance == null) return instance("enchanted");
    else return uniqueInstance;
  }
```

Create the instance using the
specified String name.

```
  public static MazeFactory instance(String name) {
    if(uniqueInstance == null)
      if (name.equals("enchanted"))
        uniqueInstance = new EnchantedMazeFactory();
      else if (name.equals("agent"))
        uniqueInstance = new AgentMazeFactory();
    return uniqueInstance;
  }
}
```

Singleton With Subclassing Method 1 (Continued)

- Problem: The instance(String) method must be modified each time a new MazeFactory subclass is added

- In Java: We could use Java class names as the argument to the instance(String) method, yielding simpler code:

```
public static MazeFactory instance(String name) {
   if (uniqueInstance == null)
      uniqueInstance = Class.forName(name).newInstance();
 return uniqueInstance;
}
```

- In C++: We could create a register of singletons.