Design patterns part 2

Noël Plouzeau

Factory Method

- Motivation
  - An interface is all is needed to request service from an object
  - A concrete class is required to create an object
    - Very often we don't have and don't need this information

Factory Method (cont'd)

- Intent
  - To provide a means for object creation that does not require selection of a concrete class

Factory Method (cont'd)

- Participants
  - Interfaces
    - Creator, Product
  - Implementation classes
    - ConcreteCreator, ConcreteProduct

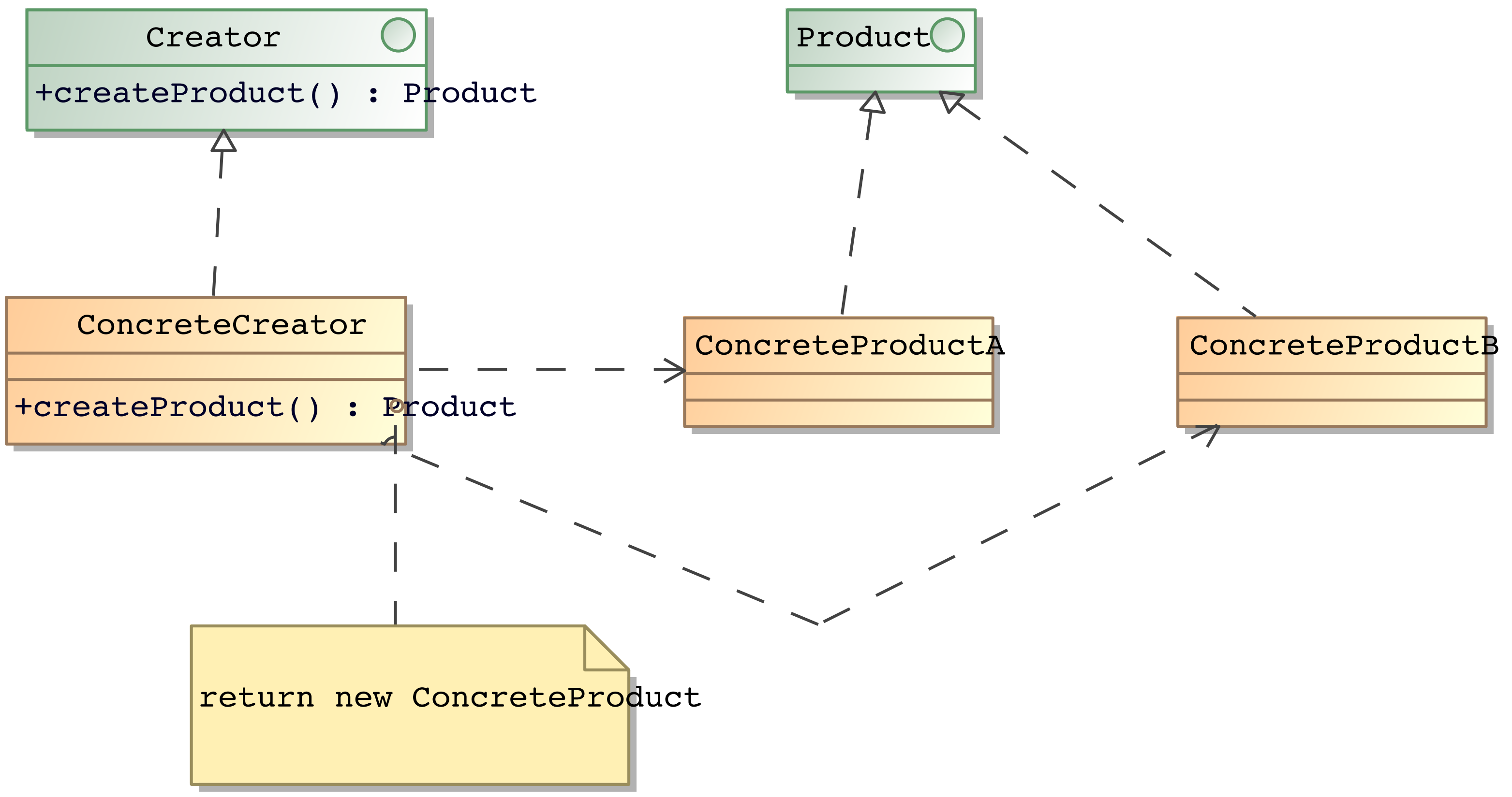
Responsibilities

- Product
  - Abstracts the class by providing service operations that the concrete object will support
- Creator
  - Defines an interface for creating products

Responsibilities (cont'd)

- ConcreteProduct
  - Implements the methods of Product
  - This is the concrete class that will be created
- ConcreteCreator
  - Implements the creation operation by doing the constructor call

Structure


Creator
+createProduct() : Product
Product
ConcreteCreator
+createProduct() : Product
ConcreteProductA
ConcreteProductB
return new ConcreteProduct

Collaborations

- The collaboration here is trivial
  - a client object (not represented in the pattern) calls a factory method
  - the implementation chooses a concrete class
  - calls the constructor using new
  - return that object

Example

```
package fr.istic.nplouzeau.factoryMethod;
import java.util.List;
/**
 * An example of application of the Factory Method design pattern
 */
public interface ListCreator {
        /**
         *  Allocates a list, with an implementation that switches
         * between ArrayList and LinkedList
         * @return      a new list object
         */
        public List<String> createList();
}
```

```
public class ListCreatorImpl implements ListCreator {
        boolean useArray = false;
        /**
         * Allocates a list, with an implementation that switches
         * between ArrayList and LinkedList
         * @return a new list object
         */

}
```

```
@Override
    public List<String> createList() {
        if (useArray) {
            useArray = !useArray; return new ArrayList<String>();
        } else {
            useArray = !useArray; return new LinkedList<String>();
        }
    }
```

The MVP design pattern

- Goal : to perform separation of concerns between the user interaction subsystem (UI) and the functional subsystem (model)
  - M = Model (functional subsystem, data plus algorithms)
  - V = View (user interaction, eg graphical user interface)
  - P = presenter (coordinates view and model)
- All view/model communication goes through the presenter

The Model role

- This role is defined by an interface
  - abstracts the implementation class or classes
  - can be implemented by a Facade design pattern
- The Model knows nothing of the View types

The View model

- This role is defined by a interface
  - As for the Model, several implementations can be defined, or aggregated using a Facade design pattern
- The Model is independent of the technology used by the view
  - This is a very important point, very costly if not met

The Presenter role

- Really sticks the View and Model parts together
- Isolate them one from another
- Drawbacks
  - can become complex if the size increases
  - sees all traffic coming through

The JavaFX framework

- JavaFX supports graphical user interfaces in Java
  - Native support of the MVP design pattern
  - Views can be described using XML files
  - The JavaFX loader will instantiate controllers and bind them to the view using Java annotations

Example of JavaFX binding

- See GitHub.com/nplouzeau GLI: HighLevelBinding1
- DoubleProperties can store a double and are builtin subjects
- A property value can be recomputed automatically when one or more properties' value change
- A listener implements the Command design pattern

The Memento design pattern

- Motivation
  - in many cases one needs to save the state of an object and then restore it later
  - but at the same time encapsulation must be preserved

Solution: Memento

- Intent
  - to provide a mechanism that allow state storage
  - will masking internal state of an object
- Participants
  - Memento, Originator, Caretaker, ConcreteMemento

Responsibilities

- Memento
  - This is an empty interface (no operations)
  - Serves as a type marker, not a service access (no services provided beyond assignment and reference R/W)

Responsibilities (cont'd)

- Originator
  - has an internal state
  - is able to create and read concrete memento objects upon request to save and restore its state

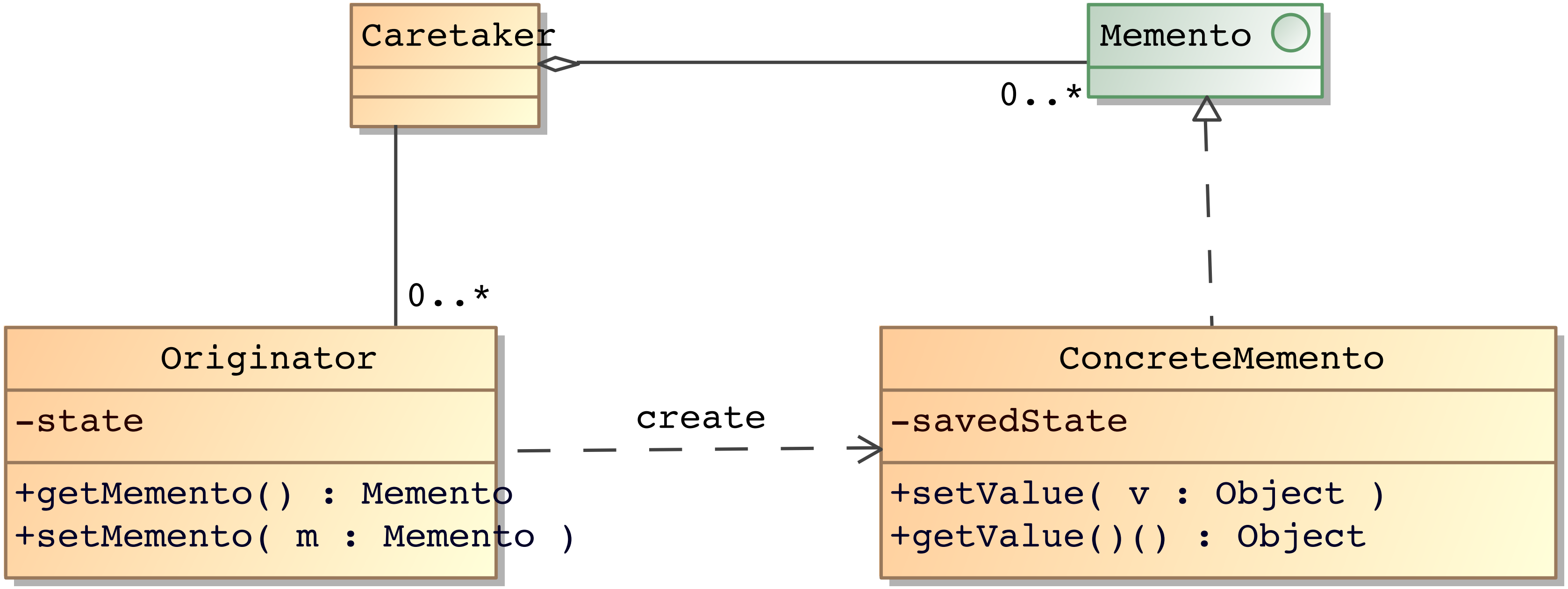
Responsibilities (cont'd)

- Caretaker
  - storage of mementos
  - asks for mementos from originator and returns them when state restoration is needed

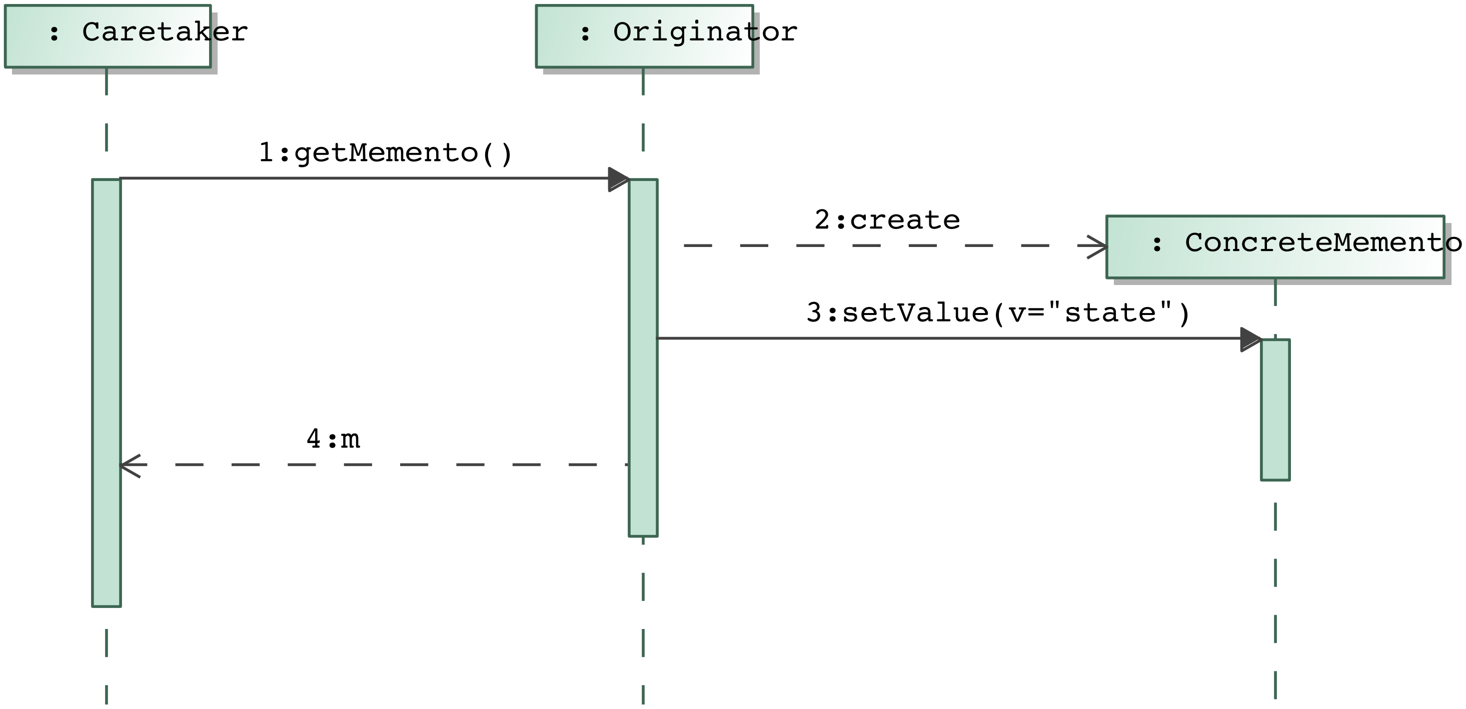
Responsibilities (cont'd)

- Concrete memento
  - contains all data necessary for restoring an originator state
  - provides accessors for the originator

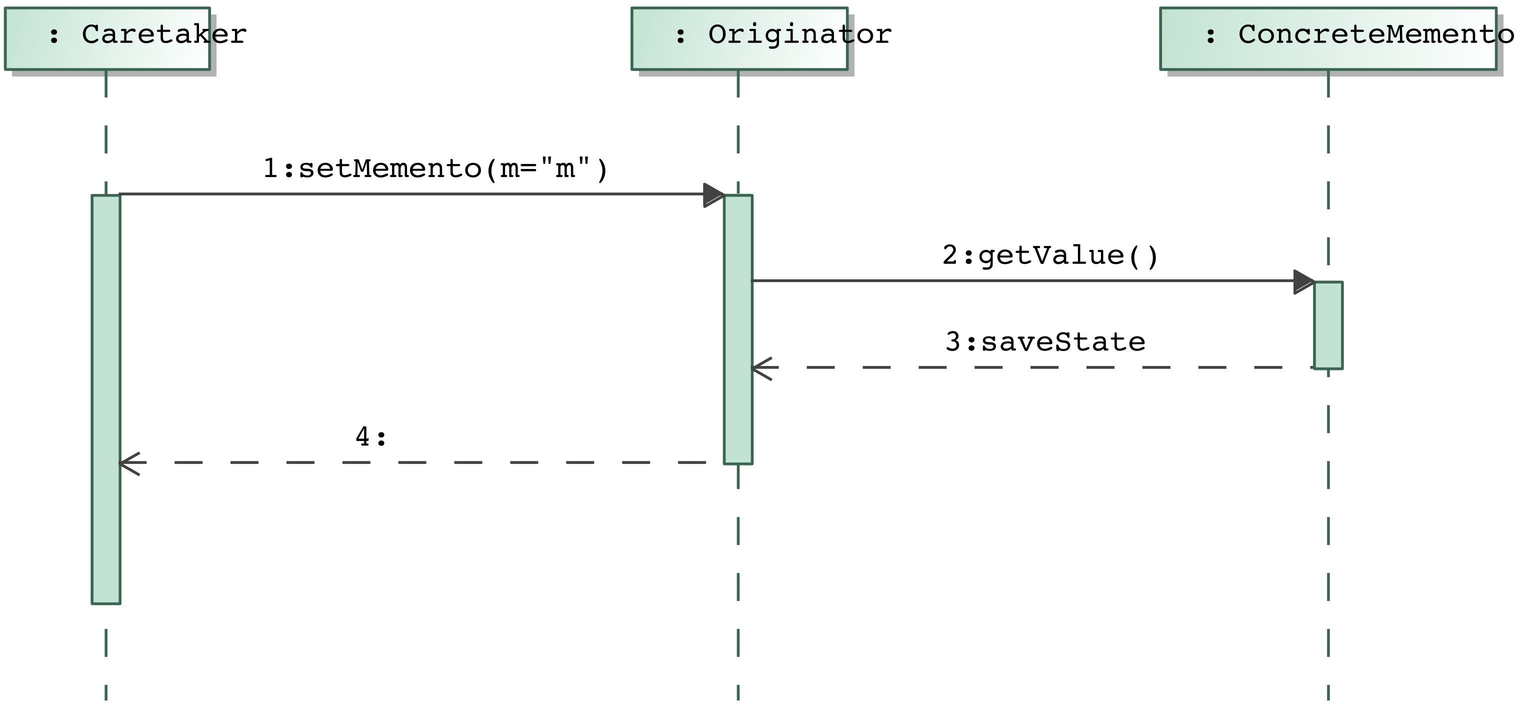
Structure


Caretaker
Memento
0..*
0..*
Originator
-state
+getMemento() : Memento
+setMemento(m : Memento)
create
ConcreteMemento
-savedState
+setValue(v : Object)
+getValue()() : Object

Collaboration: save



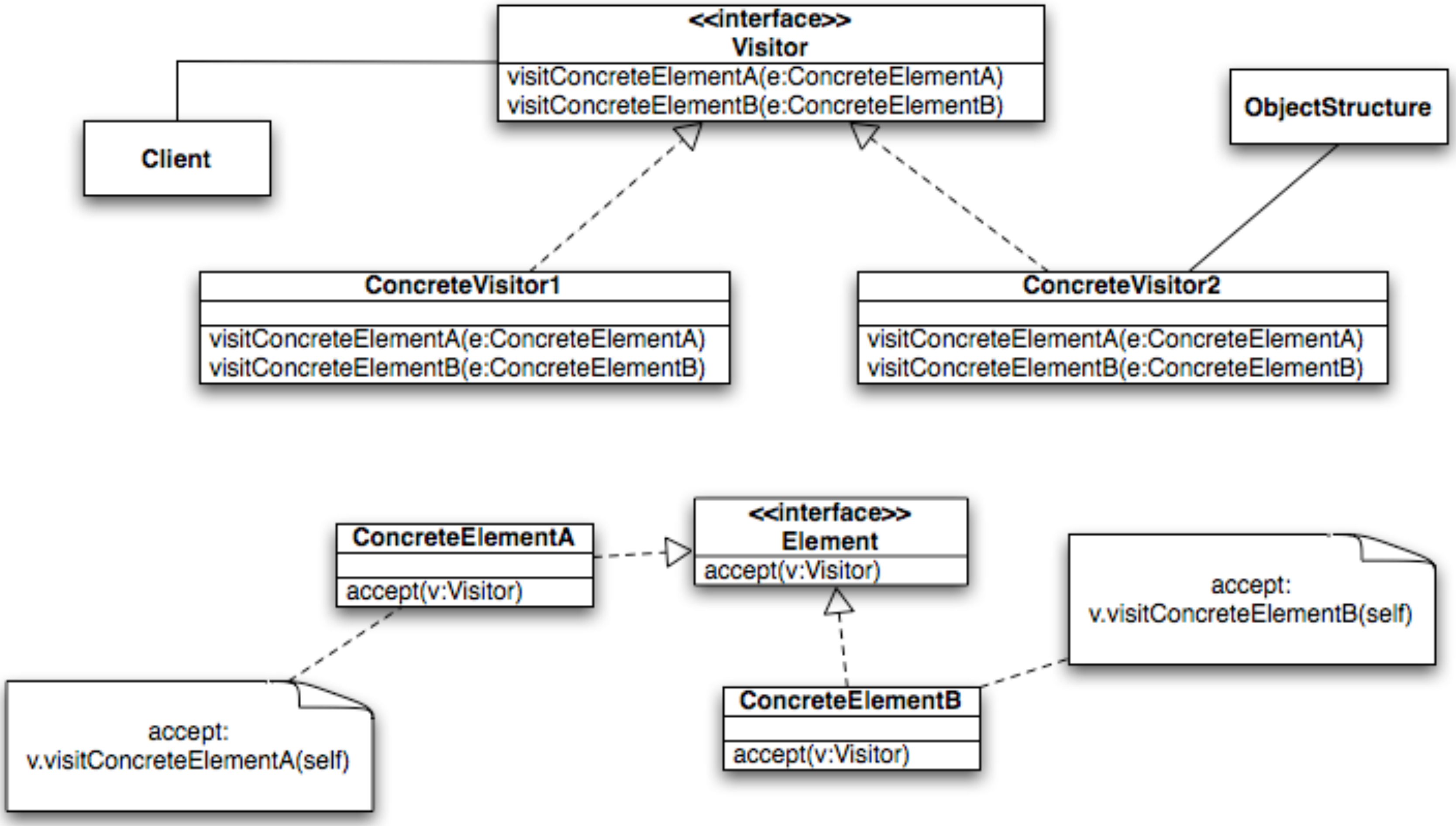
Collaboration: restore



The Visitor design pattern

- Intent
  - organize processing methods in a graph structure
  - separate types and processing
  - make addition of new algorithms easy

Structure



<<interface>>
Visitor
visitConcreteElementA(e:ConcreteElementA)
visitConcreteElementB(e:ConcreteElementB)
Client
ObjectStructure
ConcreteVisitor1
visitConcreteElementA(e:ConcreteElementA)
visitConcreteElementB(e:ConcreteElementB)
ConcreteVisitor2
visitConcreteElementA(e:ConcreteElementA)
visitConcreteElementB(e:ConcreteElementB)
ConcreteElementA
accept(v:Visitor)
<<interface>>
Element
accept(v:Visitor)
accept:
v.visitConcreteElementB(self)
accept:
v.visitConcreteElementA(self)
ConcreteElementB
accept(v:Visitor)

Responsibilities

- Visitor
  - Define a processing operation for each type of graph element
  - This operation will be called by the elements themselves
- Element
  - Define an operation that will be called by the visitor

Responsibilities (cont'd)

- ConcreteElement
  - Implementation of Element::accept()
  - To call the processing operation that is appropriate for the concrete element's type
- ConcreteVisitor
  - Implementation of a process with type specific methods

Responsibilities (cont'd)

- ObjectStructure
  - stores the elements of the graph to be visited
  - this role may be assigned to the elements themselves (use of attributes to reference neighbor nodes)
  - or it can be assigned to a specific class (e.g. Tree)

Separation of concerns

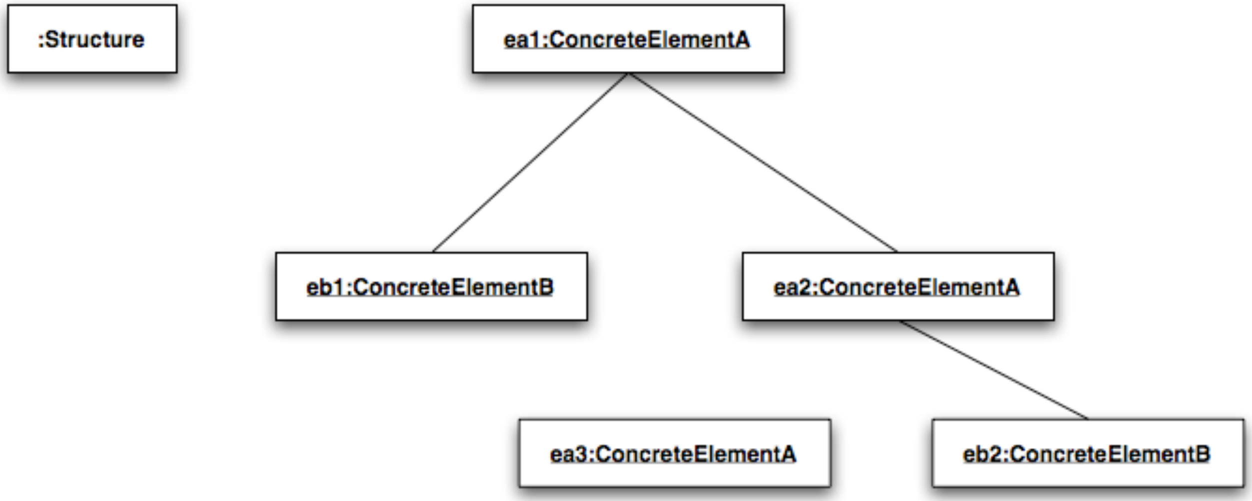
- Each concrete visitor knows what to do for each concrete element
- Each concrete element knows what to say to the visitor to trigger processing
- We have a separation between types (concrete elements) and processing (concrete visitor)

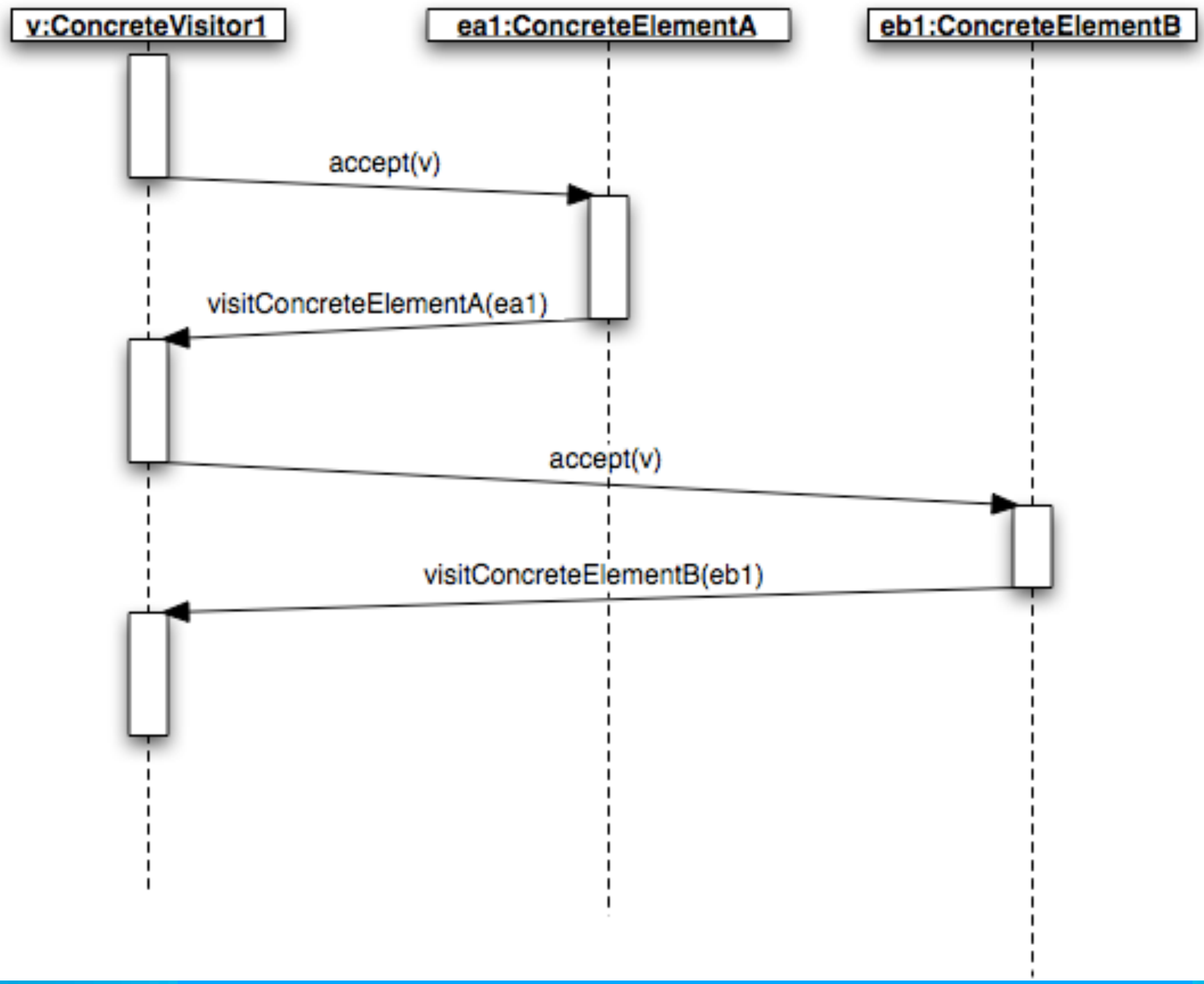
Code matrix

To compute length of a keyword

	Print	Length	Condenser
Section	1A	2A	3A
Title	1B	2B	3B
Keyword	1C	2C	3C
Document	1D	2D	3D

Object diagram




v:ConcreteVisitor1
ea1:ConcreteElementA
eb1:ConcreteElementB
accept(v)
visitConcreteElementA(ea1)
accept(v)
visitConcreteElementB(eb1)

The Builder design pattern

- Intent
  - to separate a construction algorithm from the constructed structure
- Motivation
  - a construction algorithm for a composite structure can stay the same in spite of implementation changes of the structure representation

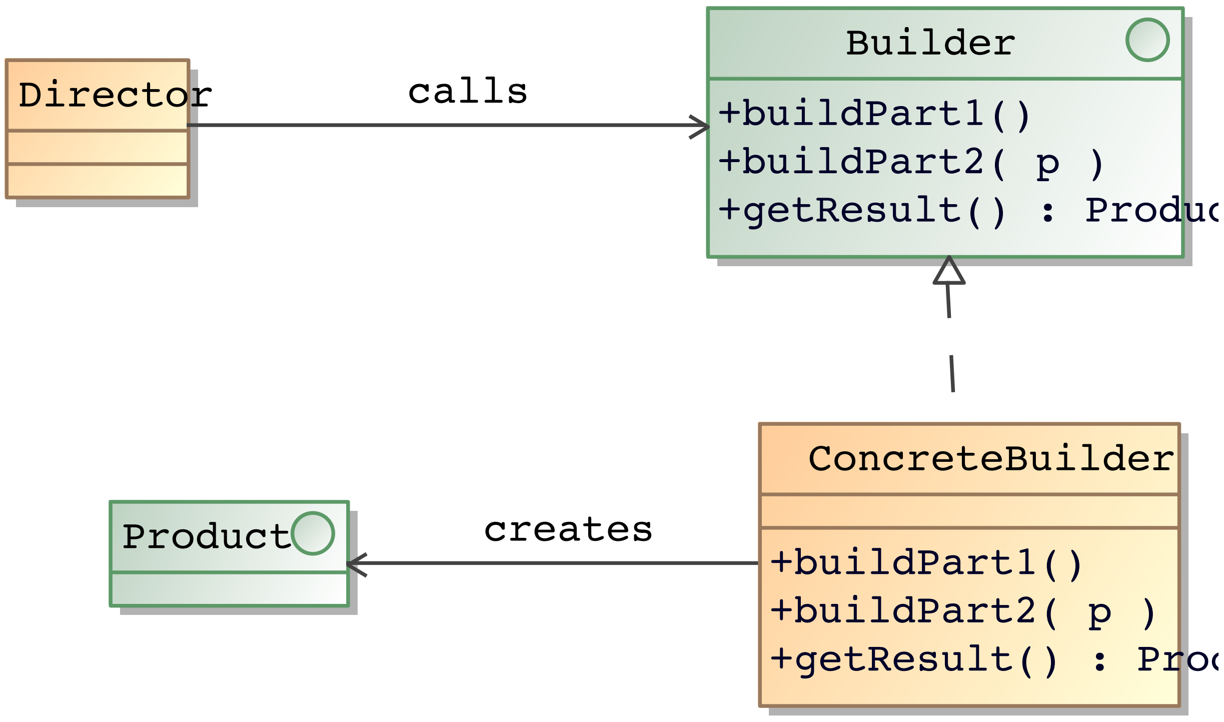
Participants of Builder

- Builder
  - Defines the operations for build parts and assemblying them
- ConcreteBuilder
  - implements Build’s operations
- Product
  - The structure under construction

Participants of Builder (cont’d)

- Director
  - The algorithm to construct the product by calling operations of creation and assembly

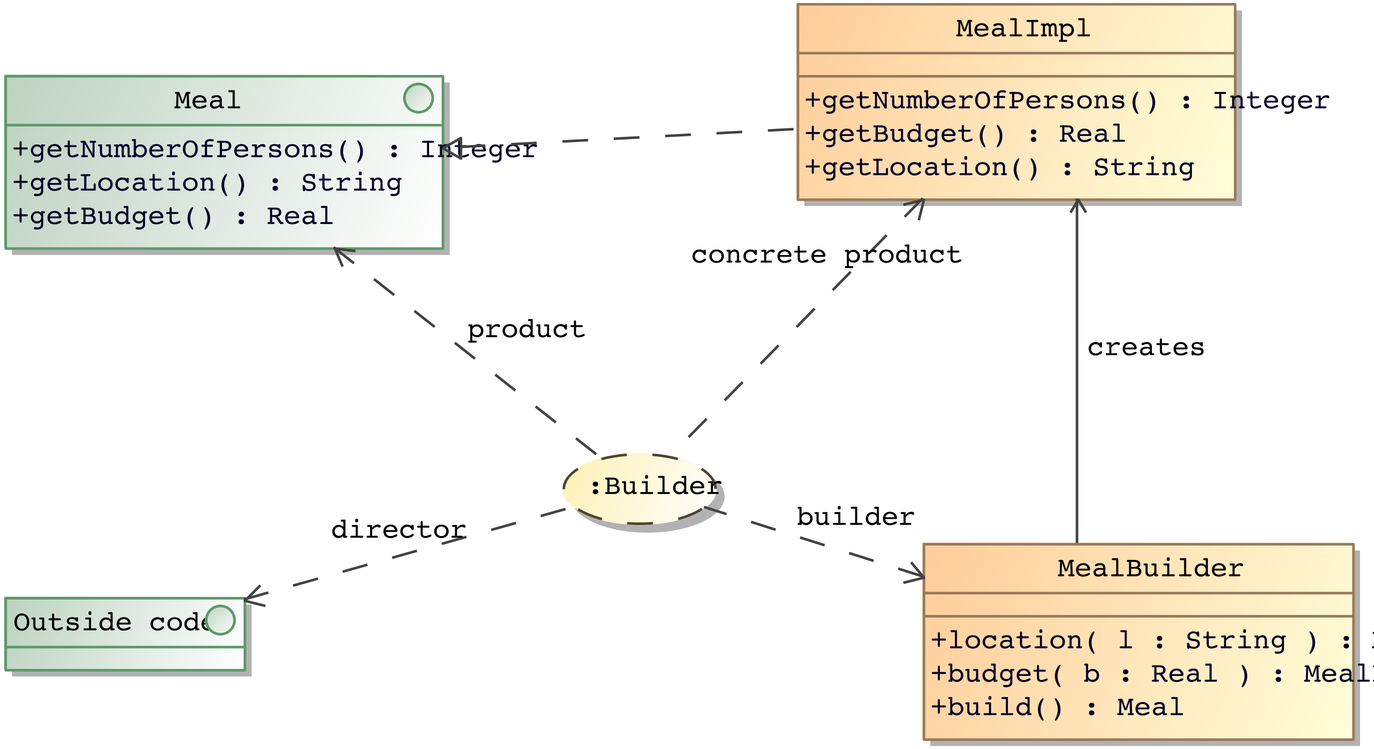
Structure


Director
calls
Builder
+buildPart1()
+buildPart2(p)
+getResult() : Produc
ConcreteBuilder
+buildPart1()
+buildPart2(p)
+getResult() : Proc
Product
creates

Example

- How to build a complex object with required and optional data
- Put the required data in the constructor
- Apply the Builder pattern to manage optional parts
- Protect construction inside the concrete builder

Example structure



Meal.java

```
package fr.istic.nplouzeau.builder;
/**
 * Author: PLOUZEAU, Noël
 * ...
 */
public interface Meal {
        public int getNumberOfPersons();
        public String getLocation();
        public double getBudget();
}
```

MealImpl.java

- Declares a public inner class as the Meal builder
- Mandatory data is passed in the builder constructor to make sure it is given
- Optional data is set by calling a data specific operation
- Final product is returned with a specific builder operation

Builder definition

```
public static class MealBuilder {
        private int numberOfPersons;      // Mandatory
        private String location = "<none defined>";        // Optional
        private double budget = 0;          // Optional
        public MealBuilder(int numberOfPersons) {
                this.numberOfPersons = numberOfPersons;
        }
```

Builder definition (cont'd)

- // Store the optional data in the builder
- public MealBuilder location(String l) {
- this.location = l;
- return this; // Allows for chaining calls
- }

Builder definition (cont'd)

- // Final construction of product
- // Builds a meal from stored parameters
- public Meal build() {
-         return new MealImpl(this);
- }

MealImpl constructor

- **private** MealImpl(MealBuilder builder) {
- numberOfPersons = builder.numberOfPersons;
- location = builder.location;
- budget = builder.budget;
- }

Example of use

- @org.junit.Before
- public void setUp() throws Exception {
- meal = new MealImpl.MealBuilder(nbPersons) .location("ISTIC").build();
- }

Example of use (cont'd)

- @org.junit.Test
- public void testGetLocation() throws Exception
- {
  Assert.assertEquals("ISTIC", meal.getLocation());
- }