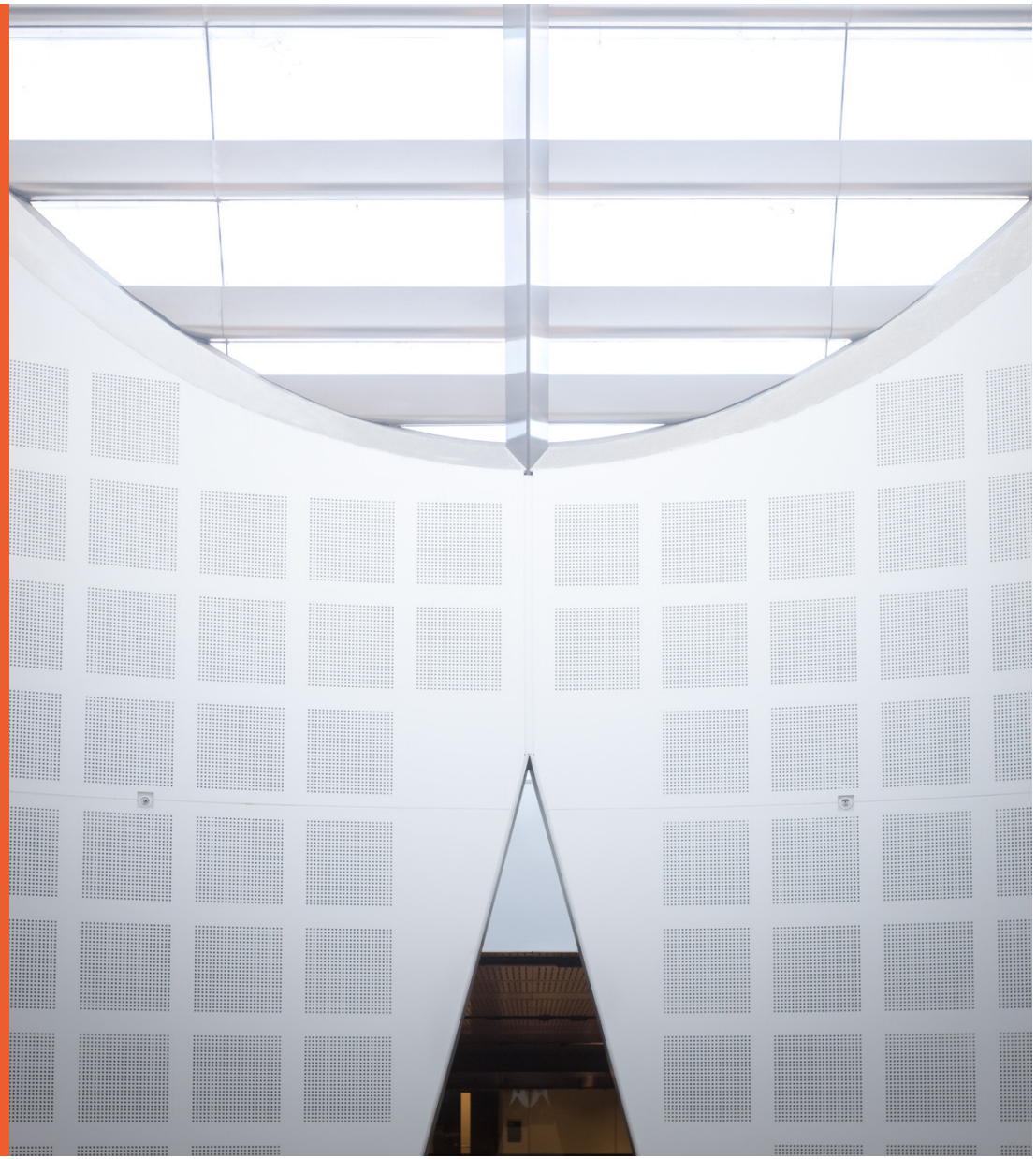
SOFT2201/COMP9201: Software Design and Construction 1

Memento and Prototype

Dr. Xi Wu
School of Computer Science

THE UNIVERSITY OF SYDNEY



Copyright warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Announcement

- Details of Assignment 3 will be released tonight (an announcement will be sent to you on Ed discussion forum once it is released)

- An announcement will be made on Ed regarding moving all on-campus tutorials held on this Thursday and this Friday to online mode because of strike

Agenda

- Behavioural Design Pattern
  - Memento

- Creational Design Pattern
  - Prototype

Behavioural Patterns (GoF)

Pattern Name	Description
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable (let algorithm vary independently from clients that use it)
State	Allow an object to alter its behaviour when its internal state changes. The object will appear to change to its class
Observer	Define a one-to-many dependency between objects so that when one object changes, all its dependents are notified and updated automatically
Memento	**Without violating encapsulation, capture and externalise an object's internal state so that the state can later be restored to this state**

Memento Design Pattern

Object Behavioural

THE UNIVERSITY OF SYDNEY

Motivated Scenario

- Suppose you are typing some words in a document, and you want to undo what you have typed just now.

Motivated Scenario – Document Perspective

```java
public class Document {
    5 usages
    private int lineNumber;
    5 usages
    private String existingContent;
    2 usages
    public String getContent() { return this.existingContent; }
    2 usages
    public void setContent(String content) { this.existingContent = content; }
    2 usages
    public int getLineNumber() { return this.lineNumber; }
    2 usages
    public void setLineNumber(int number) { this.lineNumber = number; }
    1 usage
    public void getInitState() {...}
    1 usage
    public void afterTyping() {...}
    3 usages
    public void StateDisplay() {...}
}
```

```java
public void getInitState() {
    this.lineNumber = 1;
    this.existingContent = "Hello World";
}
1 usage
public void afterTyping() {
    this.lineNumber = 2;
    this.existingContent = "Hello World \nHello Programming";
}
3 usages
public void StateDisplay() {
    System.out.println("The current line number is " + this.lineNumber);
    System.out.println("The current content is " + this.existingContent);
}
```

Motivated Scenario – Client Perspective

```
Document originalDoc = new Document();
originalDoc.getInitState();
originalDoc.StateDisplay();

Document backup = new Document();
backup.setContent(originalDoc.getContent());
backup.setLineNumber(originalDoc.getLineNumber());

originalDoc.afterTyping();
originalDoc.StateDisplay();

originalDoc.setLineNumber(backup.getLineNumber());
originalDoc.setContent(backup.getContent());
originalDoc.StateDisplay();
```

Should Client know all these details ?

Client is responsible for backup ?

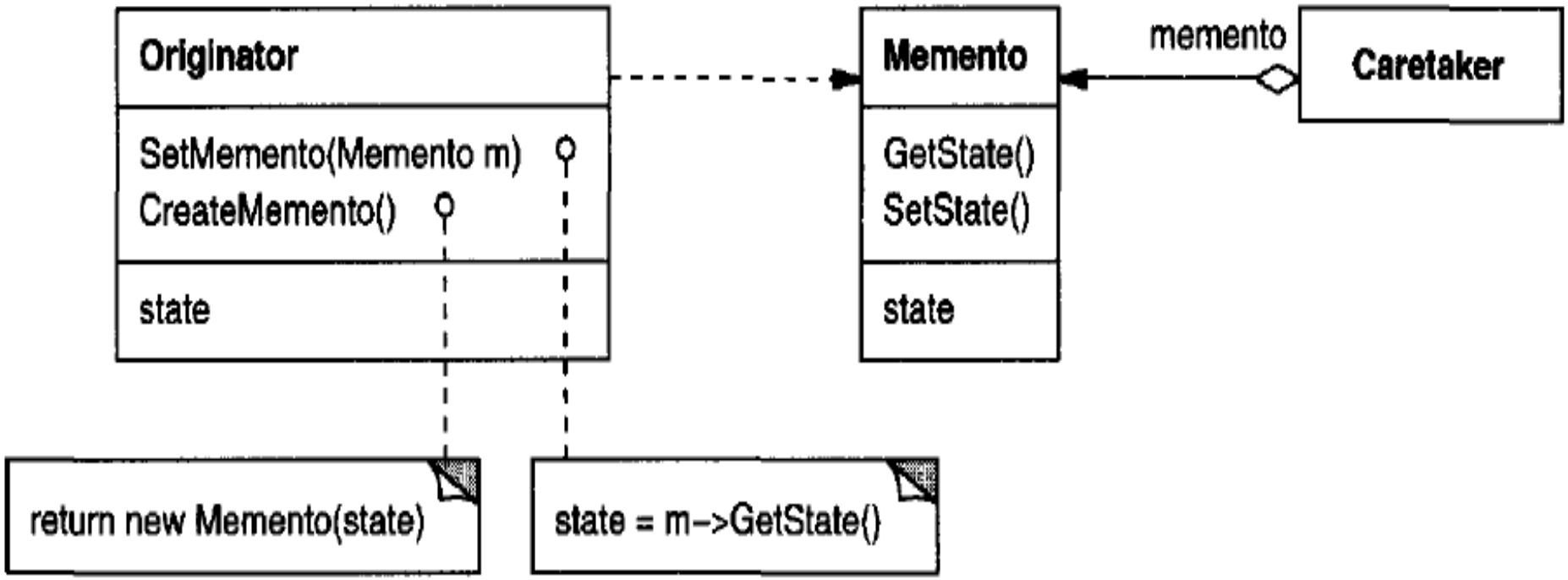
How about if we want to know the page number of the document later?

Memento Pattern

- Intent
  - Capture and externalise an object's internal state so that the object can be restored to this state later, without violating encapsulation

- Applicability
  - A snapshot of, or some portion of, an object's state must be saved so that it can be restored to that state later, and
  - A direct interface to obtaining the state would expose implementation details and break the object's encapsulation
  - Save/Load, Undo/Redo

Memento – Structure



Memento – Participants

- **Originator**
  - Creates a memento containing a snapshot of its current internal state
  - Uses the memento restore its internal state
- **Caretaker**
  - Responsible for the memento's safekeeping
  - Never operates on or examines the contents of a memento

Memento – Participants

- **Memento**
  - Stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion
  - Protects against access by objects other than the originator. Mementos have effectively two interfaces
    - *Caretaker:* sees a narrow interface to the Memento — it can only pass the memento to other objects
    - *Originator:* sees a wide interface, one that lets it access all the data necessary to restore itself to its previous state

Revisit the Motivated Example

```java
public class Originator {
    5 usages
    private int lineNumber;
    5 usages
    private String existingContent;
    1 usage
    public Memento SaveState() {
        return (new Memento(lineNumber,existingContent));
    }
    1 usage
    public void RecoverState (Memento memento) {
        this.existingContent = memento.getContent();
        this.lineNumber = memento.getLineNumber();
    }
    1 usage
    public void getInitState() {...}
    3 usages
    public void StateDisplay() {...}
    1 usage
    public void afterTyping() {...}
}
```

```java
public class Caretaker {
    2 usages
    private Memento memento;
    1 usage
    public Memento getMemento() { return memento; }
    1 usage
    public void setMemento(Memento memento) { this.memento = memento; }
}
```

```java
public class Memento {
    3 usages
    private int lineNumber;
    3 usages
    private String existingContent;
    1 usage
    public Memento (int number, String content){
        this.existingContent = content;
        this.lineNumber = number;
    }
    1 usage
    public String getContent() { return this.existingContent; }
    public void setContent(String content) { this.existingContent = content; }
    1 usage
    public int getLineNumber() { return this.lineNumber; }
    public void setLineNumber(int number) { this.lineNumber = number; }
}
```

Revisit the Motivated Example – Client Perspective

```
Originator originalDoc = new Originator();
originalDoc.getInitState();
originalDoc.StateDisplay();

Caretaker stateKeep = new Caretaker();
stateKeep.setMemento(originalDoc.SaveState());

originalDoc.afterTyping();
originalDoc.StateDisplay();

originalDoc.RecoverState(stateKeep.getMemento());
originalDoc.StateDisplay();
```

Memento – Implementing Undo

- Consider implementing checkpoints and an undo mechanism that lets users revert operations

- How could such behaviour be designed?
  - What information should be captured?
  - How and where should the information be stored?
  - Is the way you proposed to store the information a good design? Why? Why not?

Memento – Implementing Undo

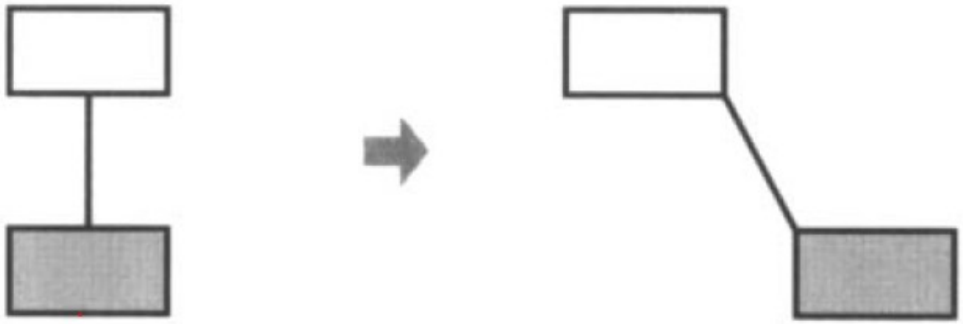
- State information must be stored so that objects can be restored to their previous state

- Encapsulation – objects normally encapsulate some or all of their state which makes it inaccessible to other objects and impossible to save externally

- Exposing this state would violate encapsulation; and can compromise application's reliability and extensibility

Memento – Graphics Editor (Undo)

- A graphics editor allow users to connect and move shapes
  - E.g., connect two rectangles with a line
  - Rectangles stay connected when the user moves either of the rectangles
  - The editor ensures that the line stretches to maintain the connection



Memento – Graphics Editor (Undo) – Problem

- ConstraintSolver object records connections as they are made and generates mathematical equations that describe them
- ConstraintSolver uses the results of its calculation to rearrange the graphics so that they maintain the proper connections
- How to "undo" a move operation?
  - Store the original distance moved and move the object back to an equivalent distance

Memento – Graphics Editor (Undo) – Problem

- Does the ConstraintSolver's public interface suffice to allow precise reversal of its effect on other objects?
  - The undo mechanism must work closely with the ConstraintSolver to re-establish the previous state
  - Should avoid exposing the ContraintSolver's internals to the undo mechanism

- How to solve this problem?

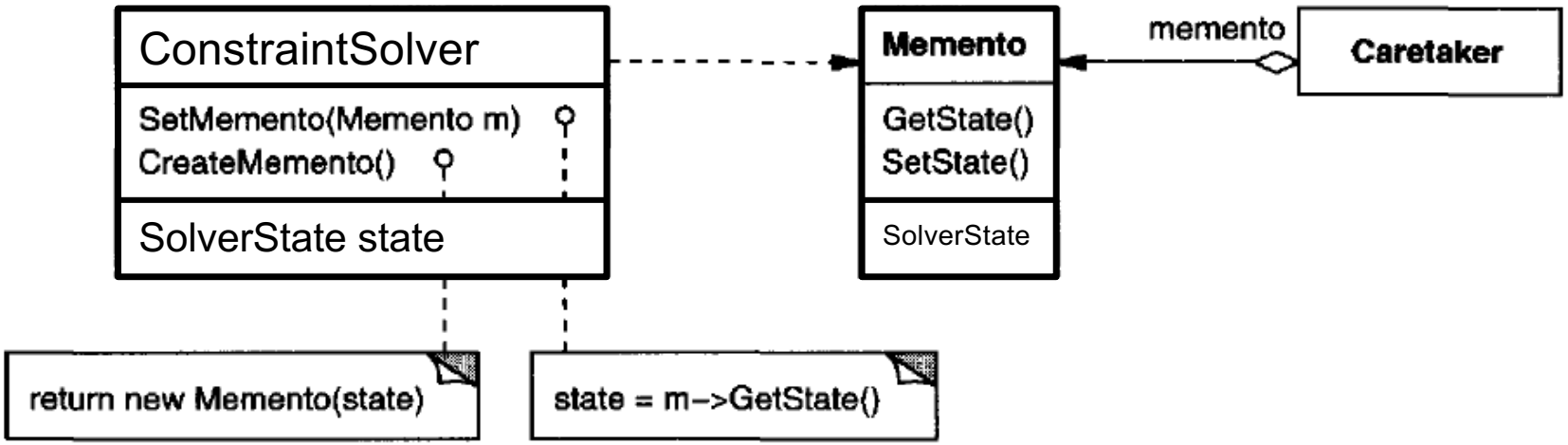
Memento – Graphics Editor (Undo) – Solution

- The undo mechanism requests a memento from the originator when it needs to checkpoint the originator's state
- The originator initialise the memento with current state information
- Only the originator can store and retrieve information from memento – the memento is not transparent to other objects

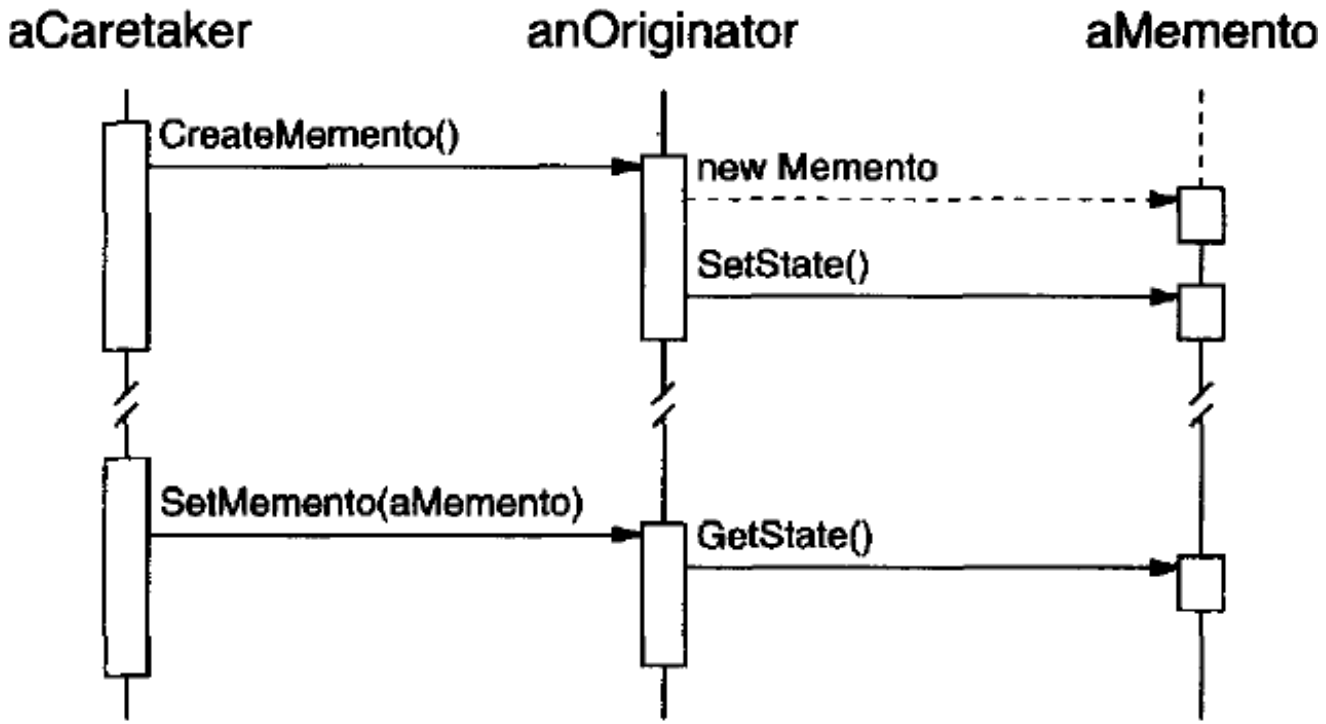
Memento – Graphics Editor (Undo) – Solution

1. When a user makes a move operation, the editor requests a memento from the ConstraintSolver
2. The ConstraintSolver creates and returns a memento, an instance of a class (e.g., SolverState), which contains data structures that describe the current state of the ConstraintSolver's internal equations and variables
3. When the user undoes the move operation, the editor gives the SolverState back to the ConstraintSolver
4. The ConstraintSolver changes its internal structures to return its equations and variables to their exact previous state

Memento – Structure



Memento – Collaborations



Memento – Consequences (1)

- Preserve encapsulation boundaries
  - By protecting other objects from potentially complex originator internals
- Simplifies Originator
  - Originator keeps the versions of internal state that clients have requested
- Might be expensive
  - If the originator must copy large amounts of information to store in the memento or,
  - If clients often create and return mementos to the originator

Memento – Load/Save

Text document editor

- We want to save the state so we can load it again later
- The Originator contains the state that represents the document being edited (and perhaps the interface to the editing functionality.)

Memento – Load/Save

Text document editor

- We want to save the state so we can load it again later
- The Originator contains the state that represents the document being edited
- A Memento stores the state, and has an interface that gives an Originator full access to the state (for restoring purposes).
- If a Caretaker needs to store a state, it asks the Originator for a Memento representing the current state, then stores it.
- If a Caretaker needs to restore a state, it asks the Originator to restore state, and passes it a Memento containing the state.

Memento – Load/Save

Text document editor

- Originator
  - Knows what the state is and how to use it
- Memento
  - Knows of state, but only needs to store it and restrict access
- Caretaker
  - Knows how to store state, and how to get/set it from the Originator.

Creational Patterns (GoF)

Pattern Name	Description
Factory Method	Define an interface for creating an object, but let sub-class decide which class to instantiate (class instantiation deferred to subclasses)
Builder	Separate the construction of a complex object from its representation so that the same construction process can create different representations
Prototype	**Specify the kinds of objects to create using a prototype instance, and create new objects by copying this prototype**
Singleton	Ensure a class only has one instance, and provide global point of access to it

Prototype Design Pattern

Object Creational

THE UNIVERSITY OF SYDNEY

Motivated Scenario

- Suppose you are making certificate to a large number of students. The content of the certificate is the same whereas the personal information for each student is different.

Prototype

- Intent
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype
- Applicability
  - When a system should be independent of how its products are created, composed, and represented
  - To avoid building a class hierarchy of factories that parallels the class hierarchy of products
  - When instances of a class can have one of only a few different combinations of state
  - **When the classes to instantiate are specified at run-time**

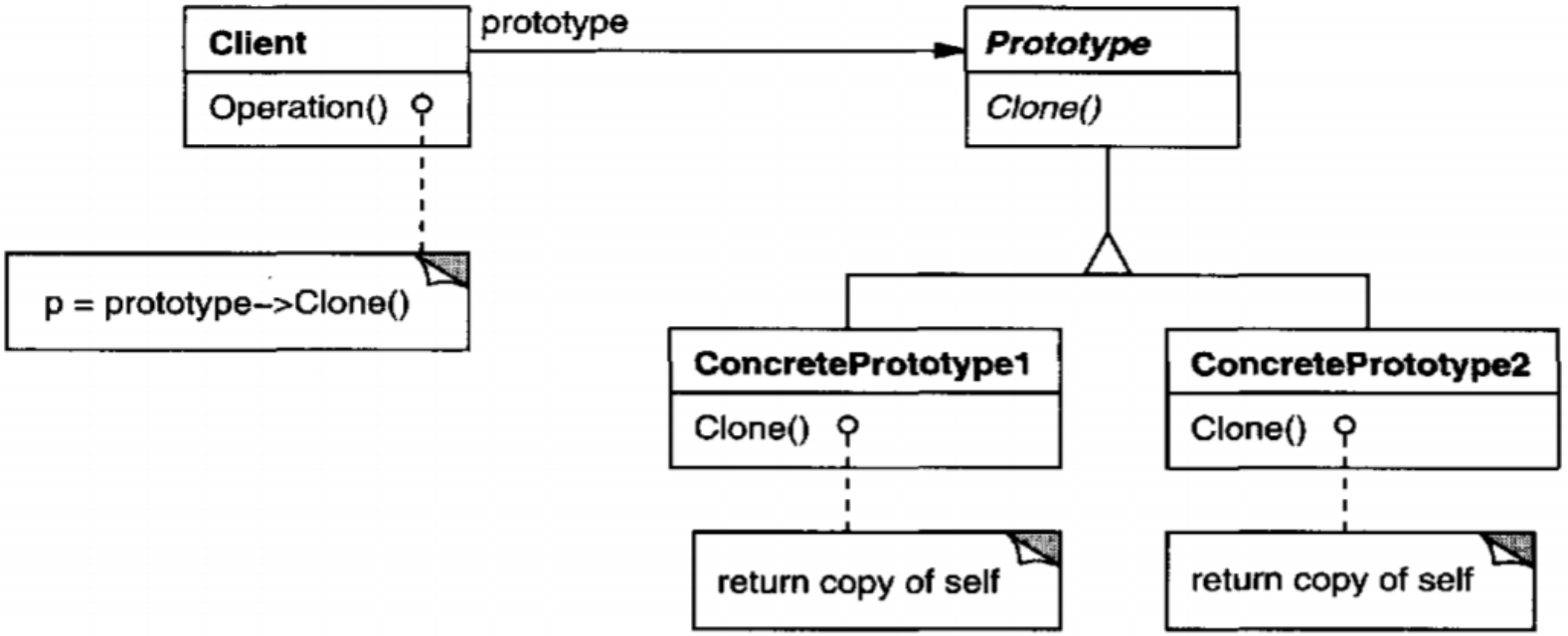
Question in Mind for Prototype

- A simple example

```
public interface Entity {}
// various classes that implement Entity
…
AbstractCollection<Entity> entities;
```

- How do you make a deep copy of your container?

Prototype – Structure



Prototype – Participants

- **Prototype**
  - Declares an interface for cloning itself
- **ConcretePrototype**
  - Implements an operation for cloning itself.
- **Client**
  - Creates a new object by asking a prototype to clone itself

- **Collaborations**
  - A client asks a prototype to clone itself

Prototype – Consequences (1)

- It hides the concrete product classes from the client, thereby reducing the number of names clients know about
- These patterns let a client work with application-specific classes without modification
- **Each subclass of prototype must implement the clone operation**

Prototype – Consequences (2)

1- Adding and removing products at run-time

- New concrete product class can be incorporated into a system
  - The client can install and remove prototypes at run-time

2- Specifying new objects by varying values

- New kinds of objects can be defined by instantiating existing classes and registering the instances as prototypes of client objects
- A client can exhibit new behaviour by delegating responsibility to the prototype – this kind of design lets users define new "classes" without programming (cloning a prototype is similar to instantiating a class)

Prototype – Consequences (3)

1- Specifying new objects by varying structure (~Builder)

- Many applications build objects from parts and subparts and for convenience let you instantiate complex, user-defined structures
  - Editors for circuit designs build circuits out of sub-circuits; specific sub-circuits can be used

- Similarly, sub-circuit can be added as a prototype to the palette of available circuit elements
  - When the composite circuit object implements clone as a deep copy, circuits with different structures can be prototype

Prototype – Consequences (4)

4- Reduced sub-classing

- By cloning prototype instead of asking factory method to make a new object
  - Factory method produces a hierarchy of Creator classes that parallel the class hierarchy

Prototype – Consequences (4)

4- Reduced sub-classing

- By cloning prototype instead of asking factory method to make a new object

```
public class NotQuiteAFactoryMethodButSortaIs {
    private MyPrototypeInterface prototype;
    public NotQuiteAFactoryMethodButSortaIs(
            MyPrototypeInterface prototype) {
        this.prototype = prototype;
    }
    public MyPrototypeInterface create() {
        return prototype.clone();
    }
}
```

Prototype – Consequences (5)

5- Configure an application with classes dynamically

An application that wants to create instances of a dynamically loaded class won't be able to reference its constructor statically – how to do this?

- The run-time environment creates an instance of each class automatically when it's loaded, and it registers the instance with a prototype manager
- Then the application can ask the prototype manager for instances of newly loaded classes, classes that weren't linked with the program originally

Prototype – Implementation (1)

Using a prototype manager

- Keep a **registry** (**prototype manager**) of available prototypes when prototypes in a system can be created and destroyed dynamically
  - Clients will store and retrieve prototypes from the register, but will not manage them
  - Before cloning, a client will ask the register for a prototype
  - Prototype manager has operations for registering a prototype under a key and unregistering it
  - Clients can change or even browse the registry at run-time

Prototype – Implementation (2)

Implementing the clone operation

- Shallow copy vs deep copy
  - Does cloning an object in turn clone its instance variables or do the clone and original just share the variables
- Cloning with complex structures usually requires a deep copy because the clone and the original must be independent
  - Ensure the clone's component are clones of the prototype's components

Prototype – Object.clone()

Java provides a clone() method

- Is it helpful?

  Creates and returns a copy of this object. The precise meaning of "copy" may depend on the class of the object. The general intent is that, for any object x, the expression:
  x.clone() != x
  will be true, and that the expression:
  x.clone().getClass() == x.getClass()
  will be true, but these are not absolute requirements. While it is typically the case that:
  x.clone().equals(x)
  will be true, this is not an absolute requirement.

Prototype – Object.clone()

Java provides a clone() method

- Is it helpful?

“Creates and returns a copy of this object. The precise meaning of "copy" may depend on the class of the object. The general intent is that, for any object x, the expression:
...”

Ambiguous and restrictive
Deep or shallow?
Use only if you have read the api documentation, understand the limitations, and are sure the rest of the related code will behave properly.

Prototype – MyClass.copy()

Write your own clone() method

- Is it helpful?
  - The definition of shallow or deep is project dependent
  - Use is project dependent
  - If the method is public, document its use well

Prototype – MyClass.copy()

Write your own clone() method using a copy constructor

```
public class MyClass {
    // fields
    public MyClass(MyClass original) {
        // copy all fields from original to this
    }
    /** Clear and well written documentation */
    public MyClass copy() {
        return new MyClass(this);
    }
}
```

Prototype – MyInterface.copy()

Write your own clone() method using a copy constructor

```
public interface MyInterface {
    /** Clear and well written documentation */
    public MyInterface copy();
}

public class MyClass implements MyInterface {
    public MyClass copy() {
        // …
    }
}
```

Prototype – MyInterface.copy()

Write your own clone() method using a copy constructor

```
public class MyClass {
    /** Clear and well written documentation */
    public MyClass copy() { … }
}

public class MyOtherClass extends MyClass {
}
```

Prototype – MyInterface.copy()

Write your own clone() method using a copy constructor

```
public class MyClass {
    /** Clear and well written documentation */
    public MyClass copy() { … }
}

public class MyOtherClass extends MyClass {
    public MyOtherClass copy() {
        // …
    }
}
```

Don't forget your children

Prototype

- A simple example, and a simple solution

```
public interface Entity {public Entity copy();}
// various classes that implement Entity
…
AbstractCollection<Entity> entities;
```

- How do you make a deep copy of your container?
- for (Entity entity : entities) { newContainer.add(entity.copy(); }

Task for Week 10

- Submit weekly exercise on canvas before 23.59pm Saturday
- Well organize time for assignment 3 once it is released
- Prepare and ask questions during tutorials

What are we going to learn on week 11?

- Software Testing

References

- Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.