Object-Oriented Programming
Prototype Pattern
State Pattern

CSIE Department, NTUT
Woei-Kae Chen
Prototype Pattern

Creation pattern
Prototype: Intent

- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

- Use a pointer (reference) to a base class object to create (clone) a concrete class object.

```
AbstractClass *p, *p1; // client
...
p1 = new AbstractClass; // error
...
p1 = new ConcreteClass; // ok. but...
...
p1 = p->Clone();        // prototype
```
Prototype: Motivation

```
Tool
  Manipulate()

GraphicTool
  Manipulate()

Client

Prototype:

Tool
  Manipulate()

Staff
  Draw()
  Clone()

MusicNote
  WholeNote
    Draw()
    Clone()
    return copy of self
  HalfNote
    Draw()
    Clone()
    return copy of self

p = prototype->Clone()
while (user drags mouse) {
  p->Draw(new position)
}
insert p into drawing
```
Prototype: Applicability

- When system should be independent of how its products are created, composed, and represented.
  - when the classes to instantiate are specified at run-time.
  - to avoid building a class hierarchy of factories that parallels the class hierarchy of products.
  - when instances of a class can have one of a few different combinations of state → more convenient to install a corresponding number of prototypes and clone them.
Prototype: Structure

Client
  Operation()

Prototype
  Clone()

ConcretePrototype1
  Clone()
  return copy of self

ConcretePrototype2
  Clone()
  return copy of self

p = prototype->Clone()
Prototype: Participants

- **Prototype (Graphic)**
  - declares an interface for cloning itself

- **ConcretePrototype (Staff, etc.)**
  - implements an operation for cloning itself

- **Client (GraphicTool)**
  - creates a new object by asking a prototype to clone itself
Prototype: Collaboration

A Client asks a prototype to clone itself

AbstractClass *p, *p1; // client
...
...
...
pl = p->Clone(); // prototype
Prototype: Consequences (1)

- **Hides** the *concrete* product classes from the client, reducing the number of names clients know about.
  - Adding and removing products at run-time
    - add a new concrete class into a system by simply registering a prototypical instance.
  - Specifying new objects by varying values
    - vary the values of an object’s variables and clone
  - Specifying new objects by varying structure
    - cloning a subcircuit (composed as a Composite Pattern)
  - Reduced subclassing
    - do not need a creator class hierarchy
  - Configuring an application with classes dynamically.
Each subclass of Prototype must implement the **Clone** operation

- may be difficult when the internals of a concrete subclass include objects that does not support copying
- may be difficult when the internals of a concrete subclass includes objects that have **circular** references
Prototype: Implementation (1)

- **Using a prototype manager**
  - client keeps a registry of available prototypes
    - the number of prototypes need not be fixed
    - enables clients to extend without writing code

- **Implementing the *Clone* operation**
  - shallow copy versus deep copy
    - do the clone and the original share variables?
    - C++ default copy constructor does a member-wise copy → pointers will be shared between the copy and the original

- **Initializing clones**
  - no parameter for the *Clone* operation (uniform interface)
    - reset states after clone
    - introduce an *Initialize* operation
C++ example:

class AbstractClass {
public:
    virtual AbstractClass *Clone() const; 
...
};
class ConcreteClass {
public:
    AbstractClass *Clone() const {
        return new ConcreteClass(*this);
    }
...
};
Prototype: Related Patterns

- Prototype and Abstract Factory are competing patterns.
- Designs that make heavy use of Composite and Decorator patterns often can benefit from Prototype as well.
State pattern

Behavioral pattern
State: Intent

- Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Also known as: Object for States.
An object can be in one of several different states.
State: Motivation (2)

- Example: Petri Net simulator
State: Applicability

- Use the State pattern in either of the following cases
  - An object’s behavior depends on its state, and it must change its behavior at run-time depending on its state.
  - Operations have large, multipart conditional statements that depend on the object’s state.
State: Structure (1)

State as object
Similar to XXX?

Change State

ConcreteStateA
Handle()

ConcreteStateB
Handle()
State: Structure (2)

Context
+ContextInterface()

Strategy
+AlgorithmInterface()

ConcreteStrategyA
+AlgorithmInterface()

ConcreteStrategyB
+AlgorithmInterface()

ConcreteStrategyC
+AlgorithmInterface()

Change algorithm
**State: Participants**

- **Context (TCPConnection)**
  - defines the interface to client.
  - maintains an instance of a ConcreteState subclass.

- **State (TCPState)**
  - defines an interface for encapsulating the behavior associated with a particular state of the Context.

- **ConcreteState subclasses (TCPListen, etc.)**
  - each subclass implement a behavior associated with a state of the Context.
State: Collaborations

- Context delegates state-specific requests to the current ConcreteState object.
- A context may pass itself as an argument to the State object handling the request.
- Context is the primary interface for clients. Clients can configure a context with State objects.
- Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.
State: Consequences

- It localizes state-specific behavior and partitions behavior for different states.
  - avoids switch statements (good if there are many states)
  - increase the number of classes
- It makes state transitions explicit.
  - separate objects for different states makes state transitions more explicit
  - states transitions are atomic (one variable; not several)
    → protect context from inconsistent internal states.
- State objects can be shared.
  - if State objects have no instance variables, then contexts can share a State object → Flyweight pattern
State: Implementation (1)

- Who defines the state transitions?
  - Context
    - if state transitions can be implemented entirely in the Context
  - ConcreteState
    - allow State subclasses to specify their successor state and make the transition by themselves
    - add an interface to the Context that lets State objects set the context’s current state
    - disadvantage: State subclasses have knowledge of other State subclasses \( \rightarrow \) dependency

- A table-based alternative
  - the table-driven approach focuses on defining state transitions
  - the State pattern models state-specific behavior
State: Implementation (2)

● Creating and destroying State objects.
  – Trade-off
    ● create State objects ahead of time and never destroying them
    ● create State objects only when they are needed and destroy them thereafter

● Using dynamic inheritance
  – changing the object’s class at run-time
    ● not possible in most object-oriented languages
    ● possible with Self and other delegation-based languages
State: Related patterns

- Flyweight pattern explains when and how State objects can be shared
- State objects are often Singletons
  - when ConcreteState perform state transitions
- Patterns using similar ideas (inheritance and polymorphism)
  - Command: command as object
  - Strategy: algorithm as object
  - Iterator: pointer as object
  - State: state as object
  - Composite: composite as object (with uniform interface)
  - Decorator: decorator as object (with uniform interface)
  - Proxy: proxy as object (with uniform interface)