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()

Prototype

Graphic
  Draw()
  Clone()

Staff
  Draw()
  Clone()

MusiclNote

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.
Prototype: Consequences (2)

- 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
Prototype: Implementation (2)

**C++ example:**

```cpp
class AbstractClass {
public:
    virtual AbstractClass *Clone() const;
...
};
class ConcreteClass {
public:
    AbstractClass *Clone() const {
        return new ConcreteClass(*this);
    }
...
};
```

**Copy Constructor**
Prototype: Related Patterns

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