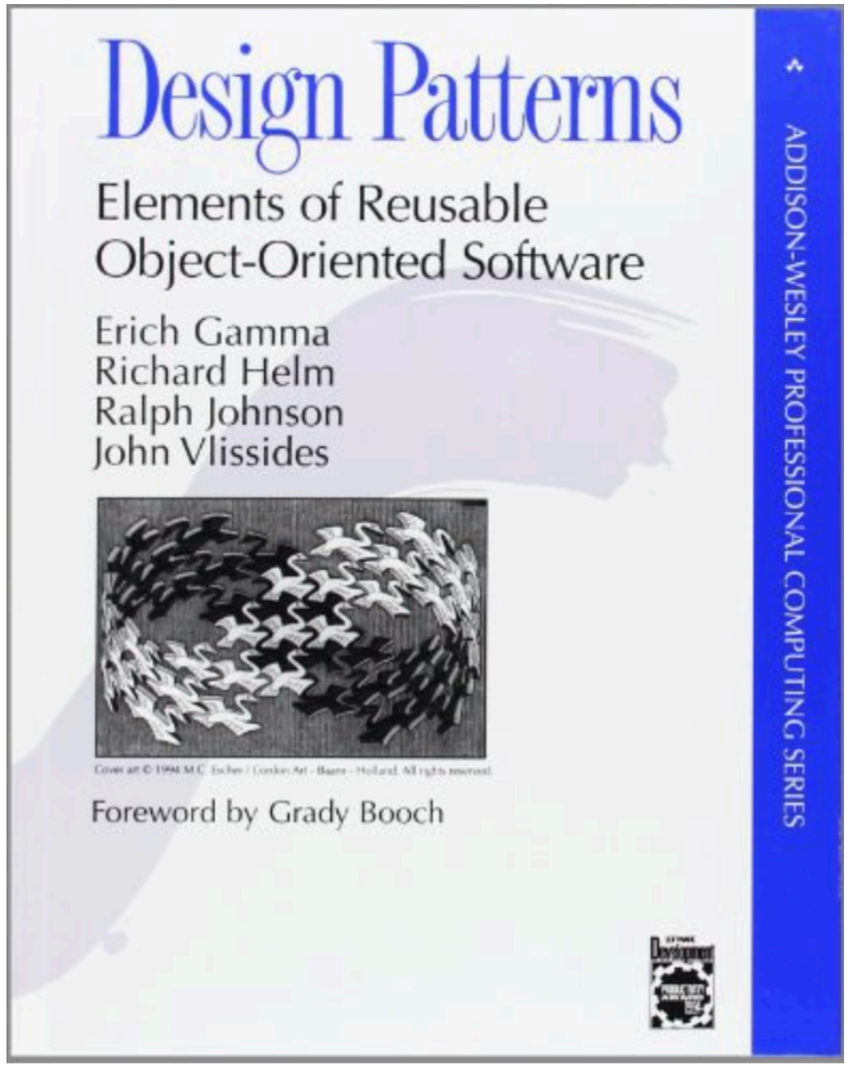
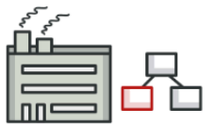
Design Pattern



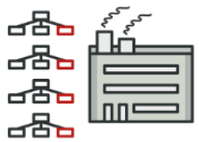
- 提出者：Erich Gamma、Richard Helm、Raplh Johnson和Jonhn Vlissides，常被称为四人帮(Gang of Four)

创建型模式

这类模式提供创建对象的机制，能够提升已有代码的灵活性和可复用性。

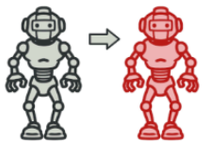


工厂方法
Factory Method

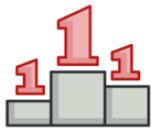


抽象工厂
Abstract Factory

生成器
Builder



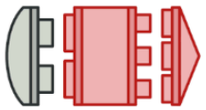
原型
Prototype



单例
Singleton

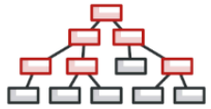
结构型模式

这类模式介绍如何将对象和类组装成较大的结构，并同时保持结构的灵活和高效。

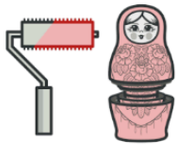


适配器
Adapter

桥接
Bridge

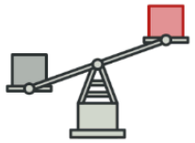


组合
Composite

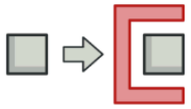


装饰
Decorator

外观
Facade



享元
Flyweight



代理
Proxy

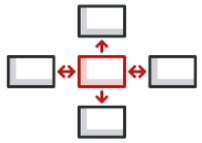
行为模式

这类模式负责对象间的高效沟通和职责委派。

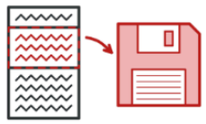
责任链
Chain of Responsibility

命令
Command

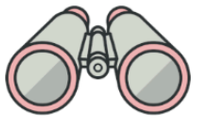
迭代器
Iterator



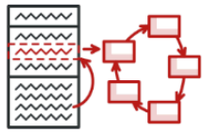
中介者
Mediator



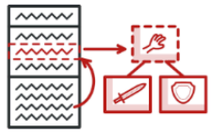
备忘录
Memento



观察者
Observer



状态
State



策略
Strategy

模板方法
Template Method

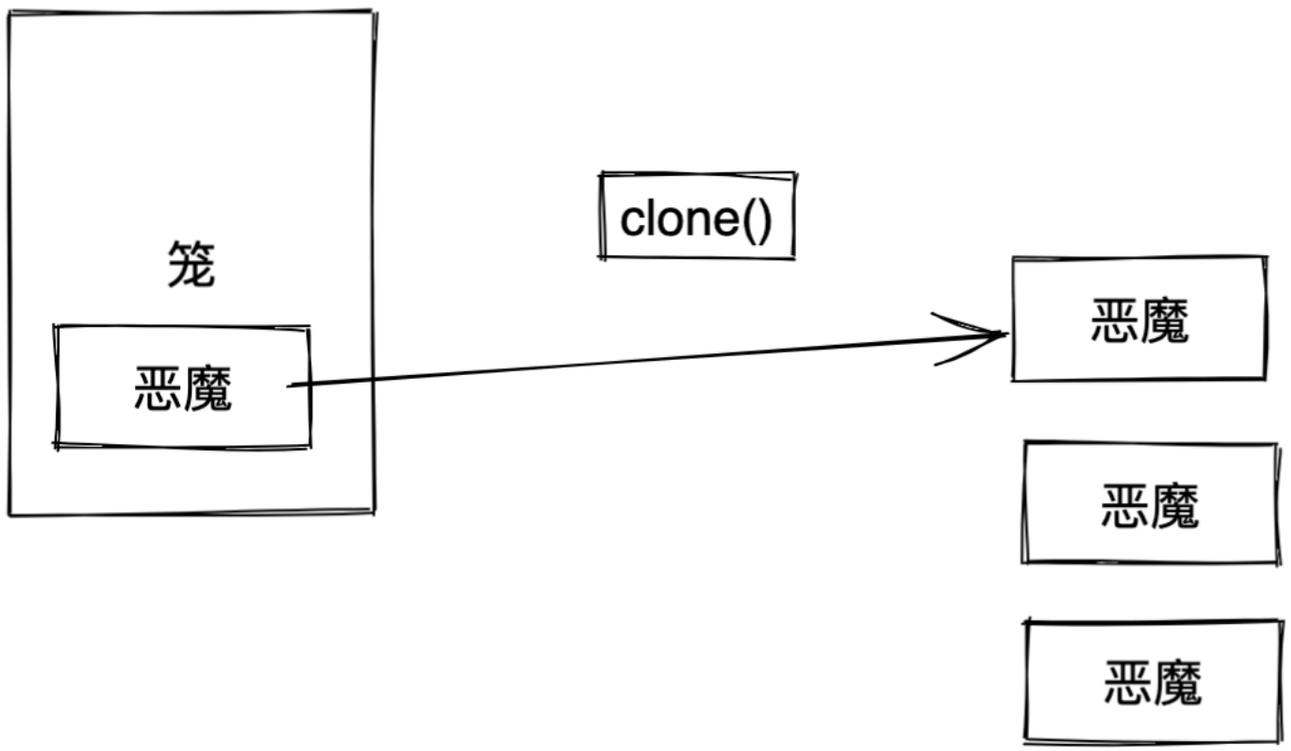
访问者
Visitor

原型模式

- Prototype
- 用原型实例指定创建对象的种类，并且通过拷贝这些原型,创建新的对象。


```
class Monster
{
  // Stuff...
};

class Ghost : public Monster {};
class Demon : public Monster {};
class Sorcerer : public Monster {};
```


```
class Spawner
{
public:
  virtual ~Spawner() {}
  virtual Monster* spawnMonster() = 0;
};

class GhostSpawner : public Spawner
{
public:
  virtual Monster* spawnMonster()
  {
    return new Ghost();
  }
};

class DemonSpawner : public Spawner
{
public:
  virtual Monster* spawnMonster()
```


笼
恶魔
clone()
恶魔
恶魔
恶魔

```cpp
class Monster
{
public:
  virtual ~Monster() {}
  virtual Monster* clone() = 0;

  // Other stuff...
};
```

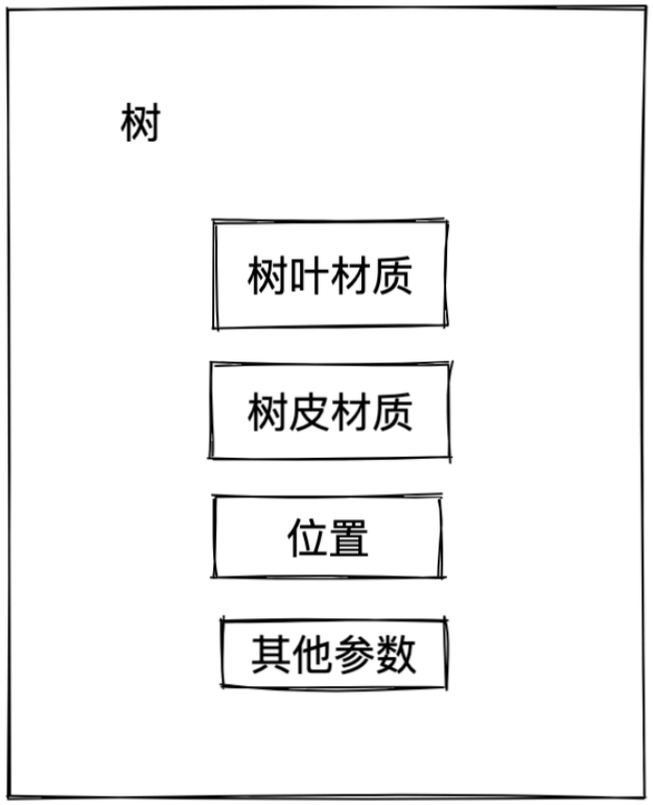
```cpp
class Ghost : public Monster {
public:
  Ghost(int health, int speed)
  : health_(health),
    speed_(speed)
  {}

  virtual Monster* clone()
  {
    return new Ghost(health_, speed_);
  }

private:
  int health_;
  int speed_;
};
```

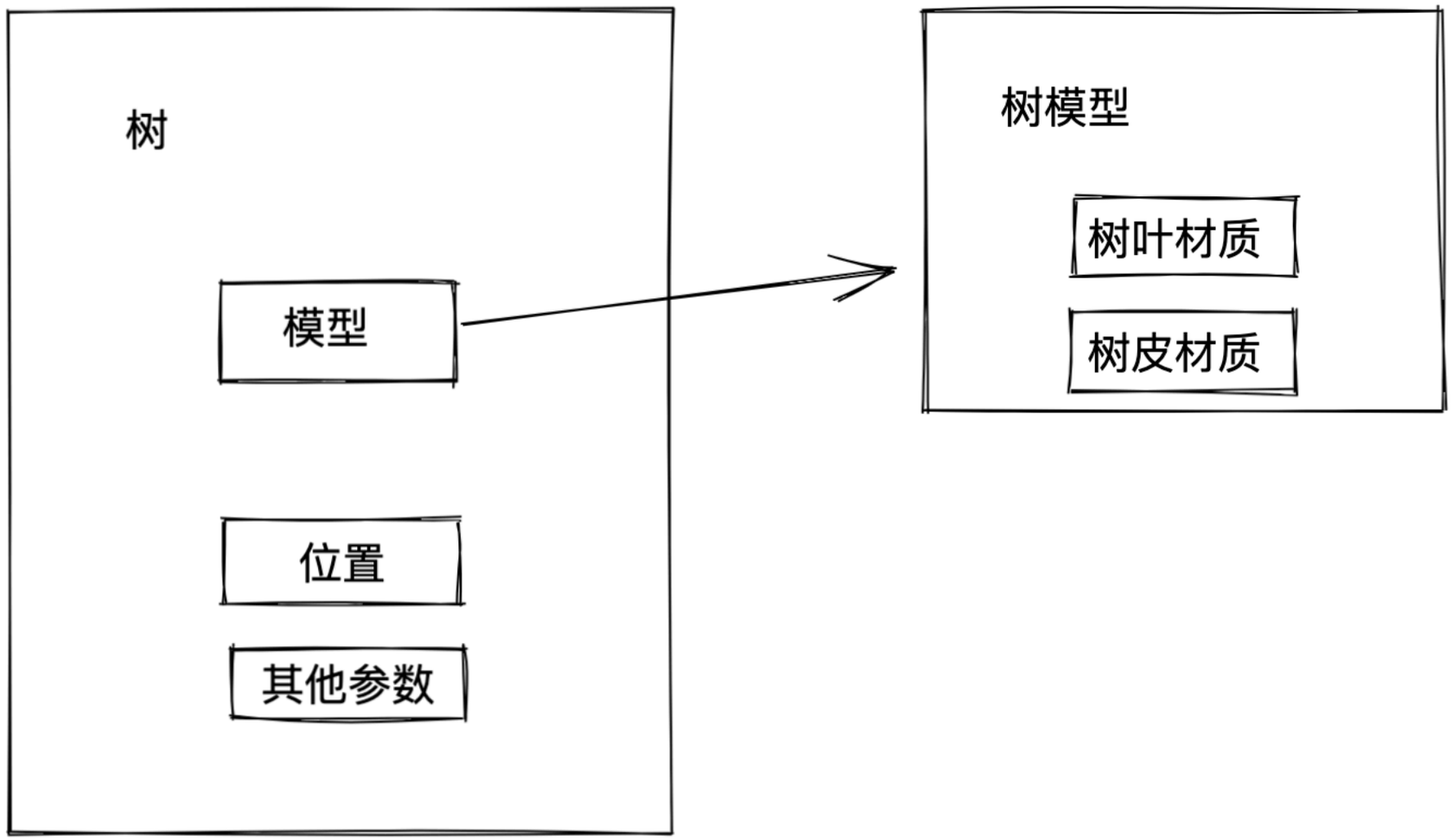
```cpp
class Spawner
{
public:
  Spawner(Monster* prototype)
  : prototype_(prototype)
  {}

  Monster* spawnMonster()
  {
    return prototype_->clone();
  }

private:
  Monster* prototype_;
};
```

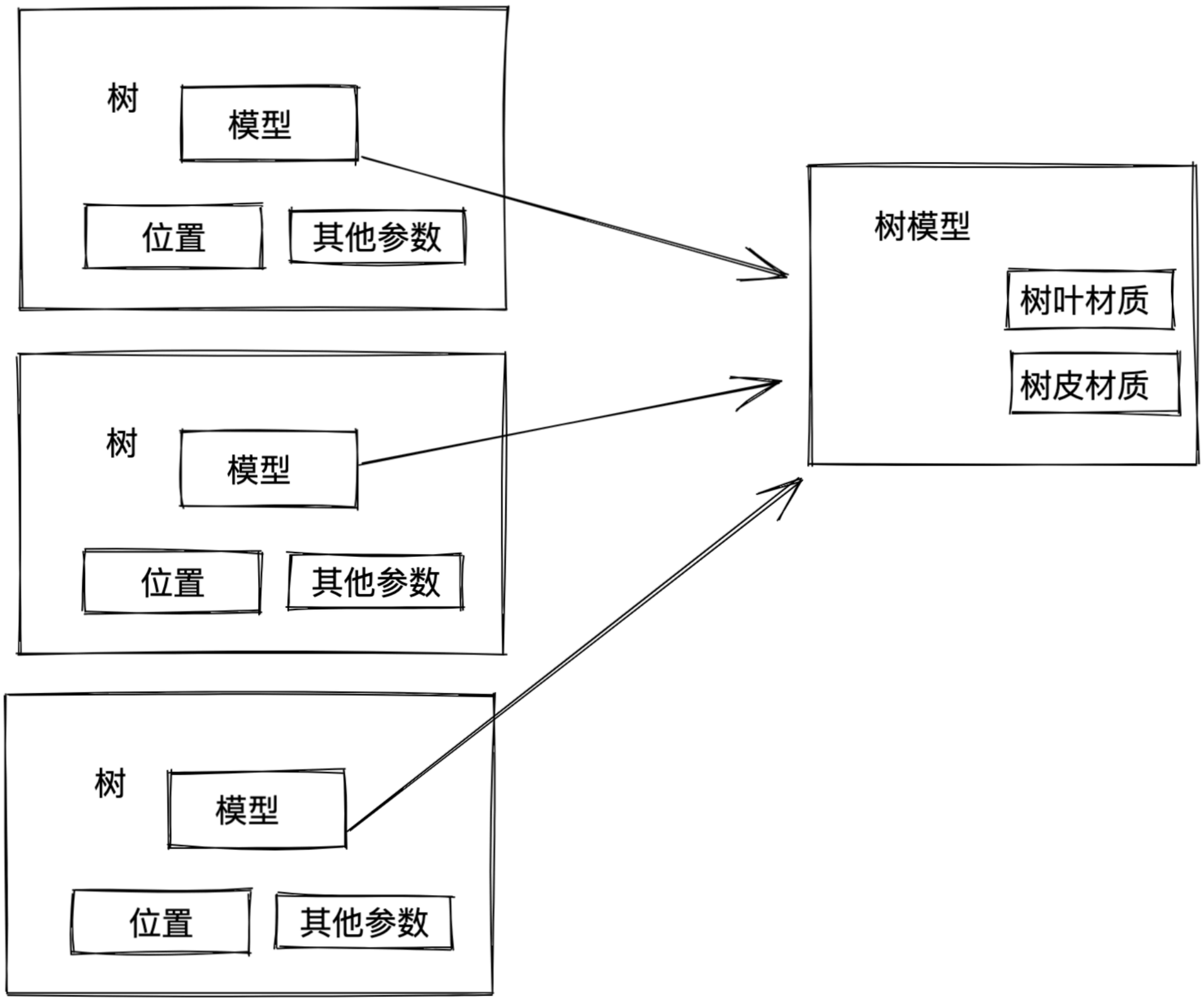
```cpp
Monster* ghostPrototype = new Ghost(15, 3);
Spawner* ghostSpawner = new Spawner(ghostPrototype);
```

享元模式

- Flyweight
- 通过共享以便有效的支持大量小颗粒对象。

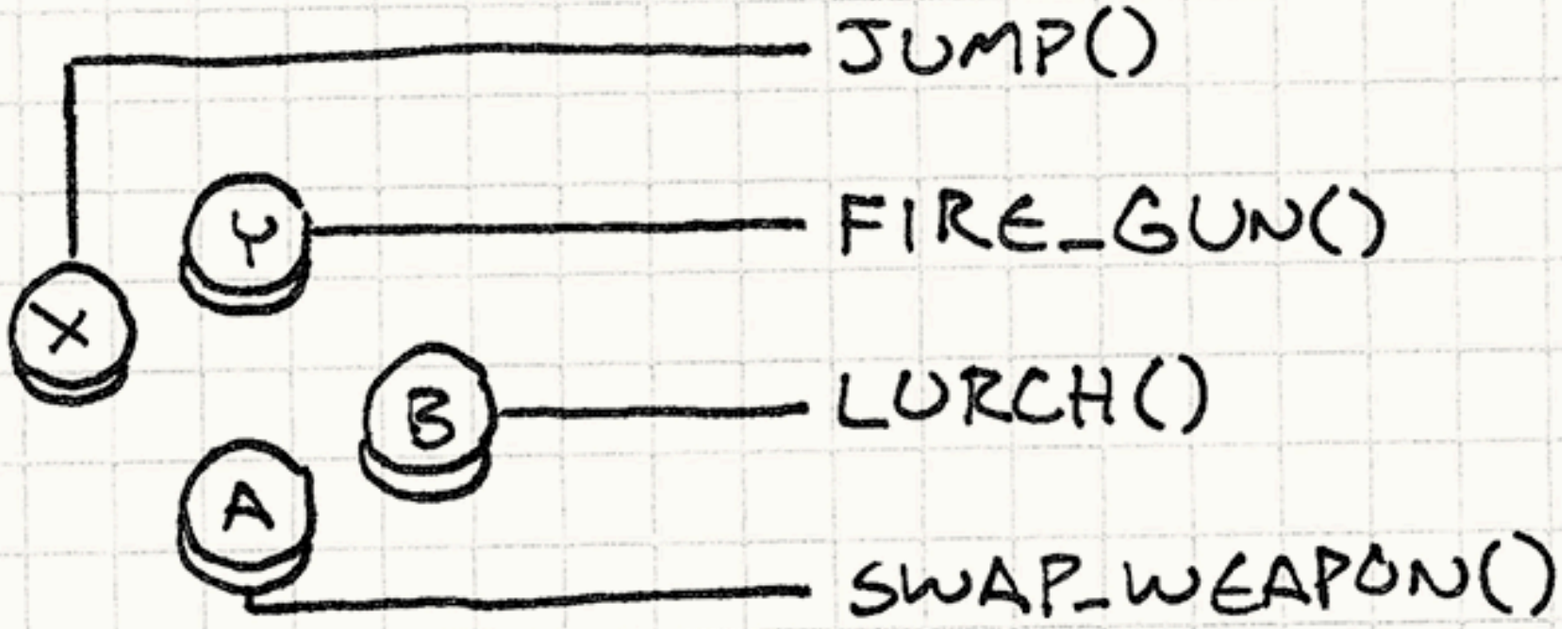

树
树叶材质
树皮材质
位置
其他参数


树
模型
位置
其他参数
树模型
树叶材质
树皮材质


树
模型
位置
其他参数
树
模型
位置
其他参数
树
模型
位置
其他参数
树模型
树叶材质
树皮材质

命令模式

- Command
- 将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可取消的操作。




```
void InputHandler::handleInput()
{
  if (isPressed(BUTTON_X)) jump();
  else if (isPressed(BUTTON_Y)) fireGun();
  else if (isPressed(BUTTON_A)) swapWeapon();
  else if (isPressed(BUTTON_B)) lurchIneffectively();
}
```

```cpp
class Command
{
public:
  virtual ~Command() {}
  virtual void execute() = 0;
};
```

```cpp
class JumpCommand : public Command
{
public:
  virtual void execute() { jump(); }
};

class FireCommand : public Command
{
public:
  virtual void execute() { fireGun(); }
};
…
```

```cpp
class InputHandler
{
public:
  void handleInput();

  // Methods to bind commands...

private:
  Command* buttonX_;
  Command* buttonY_;
  Command* buttonA_;
  Command* buttonB_;
};
```

```
void InputHandler::handleInput()
{
  if (isPressed(BUTTON_X)) buttonX_->execute();
  else if (isPressed(BUTTON_Y)) buttonY_->execute();
  else if (isPressed(BUTTON_A)) buttonA_->execute();
  else if (isPressed(BUTTON_B)) buttonB_->execute();
}
```