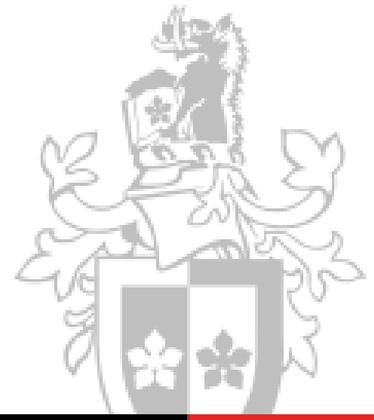
Design Patterns

by Willem van Straten and Andrew Cain



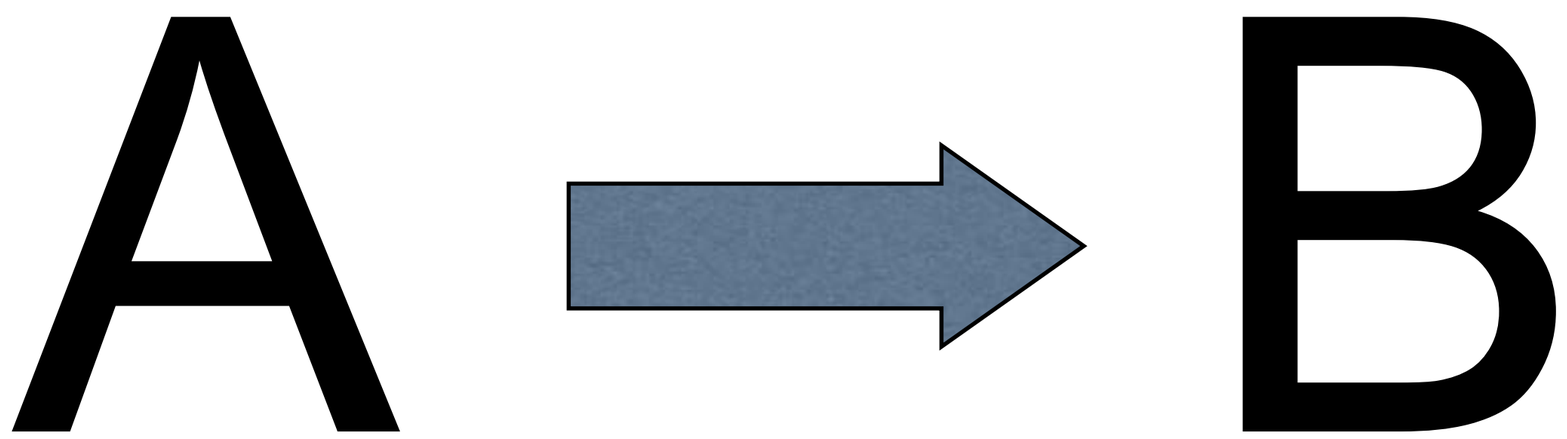
SWINBURNE UNIVERSITY OF TECHNOLOGY

Certain design issues arise repeatedly in object-oriented solutions

Many design issues have already been solved!

stackoverflow

Good developers learn from others **how** to achieve good object-oriented design



Recognize and exploit Design Patterns to increase productivity

Design patterns record solutions to commonly encountered problems

Roles and responsibilities must be factored into classes with appropriate granularity

too SMALL = too generic - does not reduce complexity
too LARGE = too problem-specific - cannot be reused

Appropriate inheritance hierarchies must be defined

Base and derived classes are strongly coupled

Appropriate delegation, collaboration and other relationships must be established

Tension and tradeoffs between reducing coupling and increasing reuse

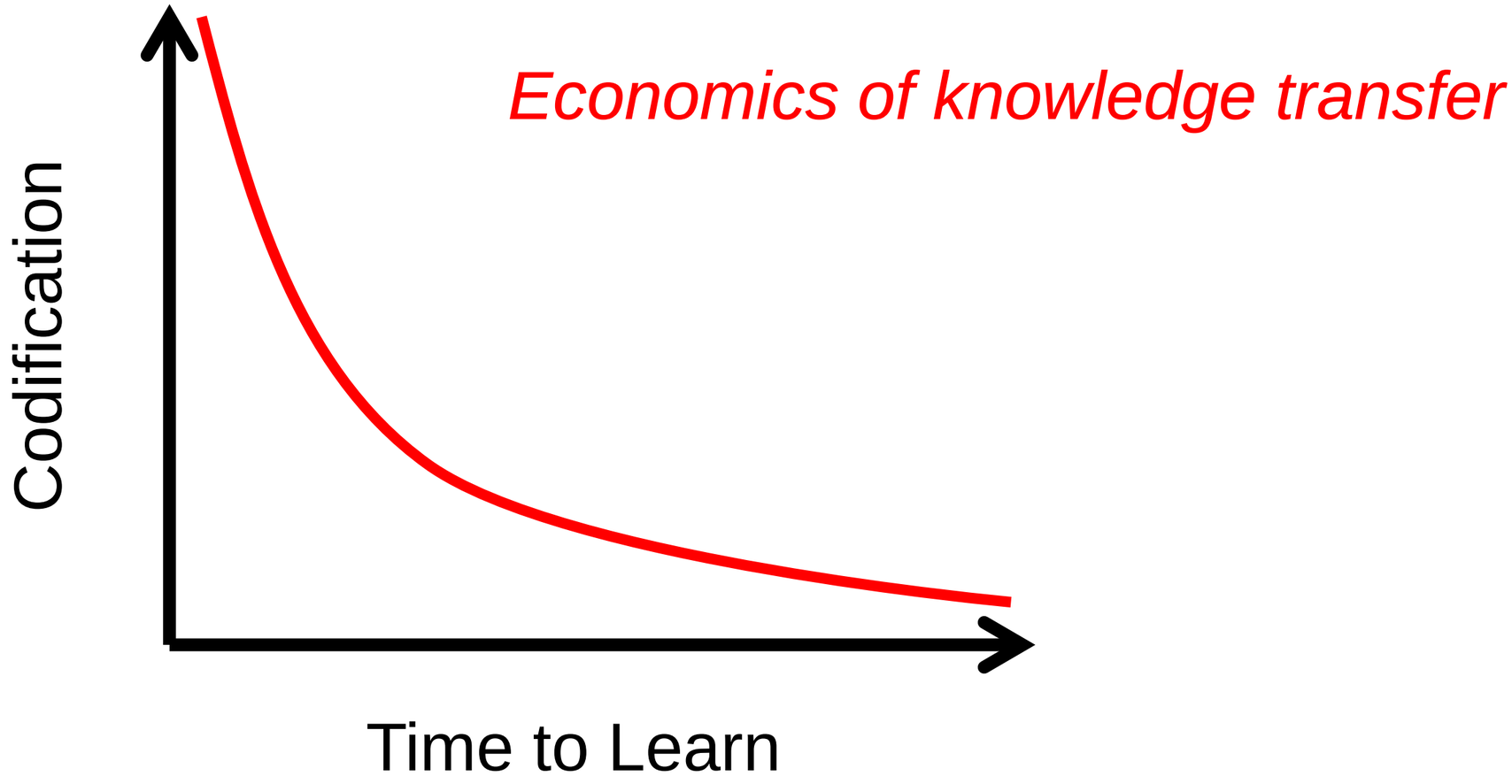
Experienced designers re-use elements of successful solutions

Recognize recurring patterns in interfaces and relationships

Reuse elements of successful designs based on prior experience

Solve problems without reinventing or rediscovering

Design Patterns codify important and recurring solutions



Design Patterns make it easier to identify abstractions and reuse proven design solutions

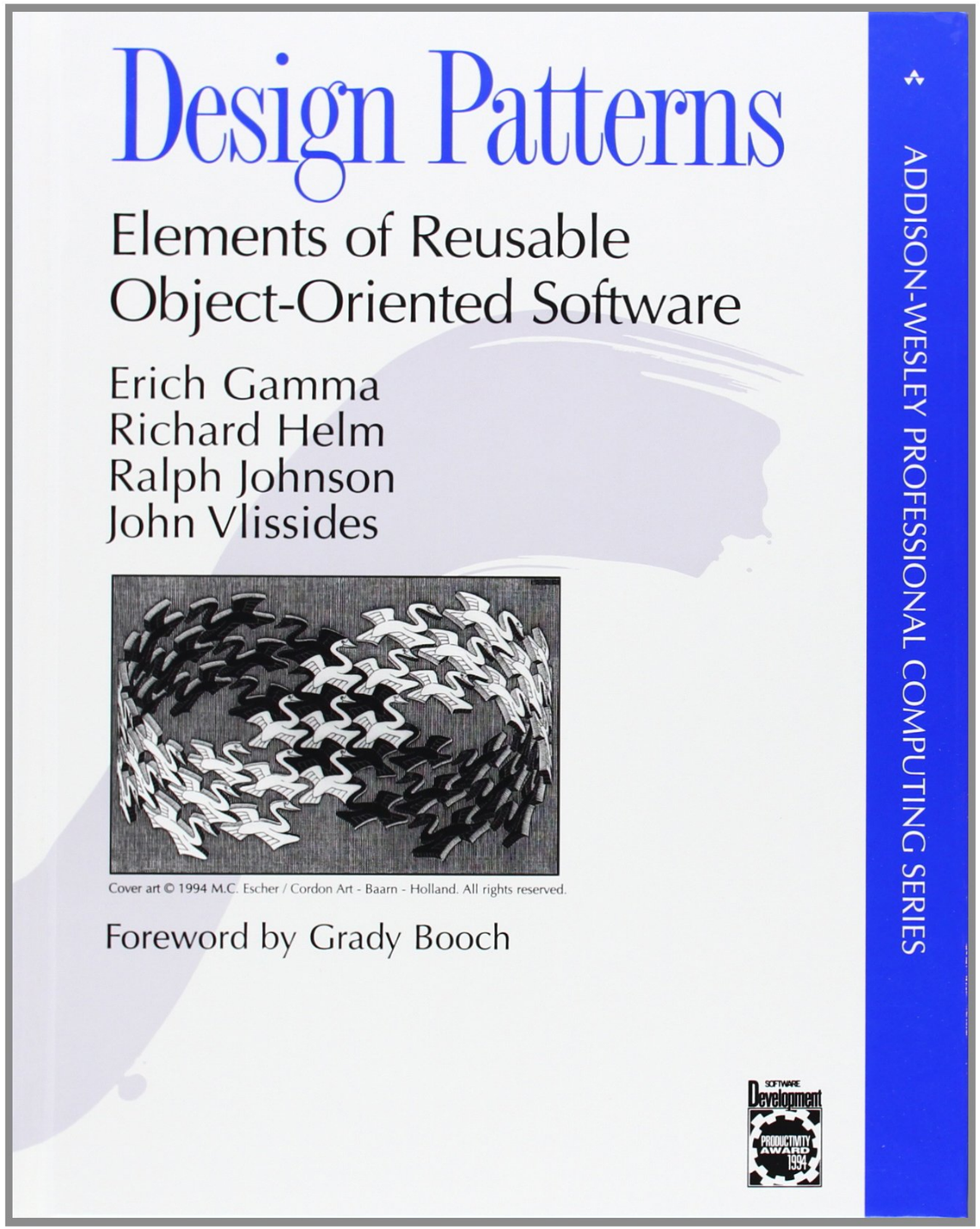
Design Patterns make proven solutions accessible to new developers

Solutions based on Design Patterns are **reusable**, **extensible**, and **maintainable**

Recognize Design Patterns to save time and effort

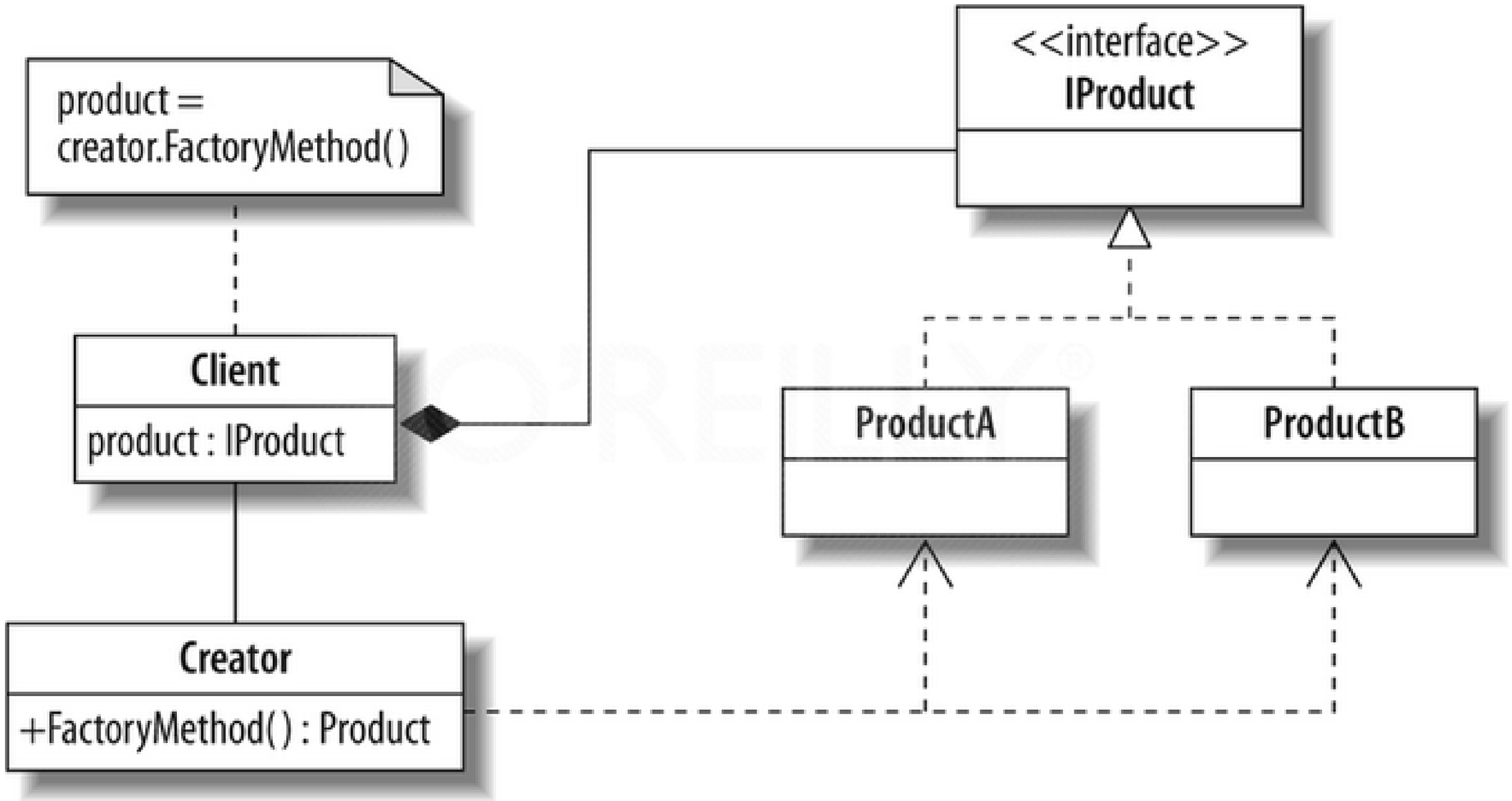
Design Patterns are broadly classified into three categories:

1. Creational – ways to create objects
2. Structural – ways to assemble objects
3. Behavioural – ways to perform common tasks



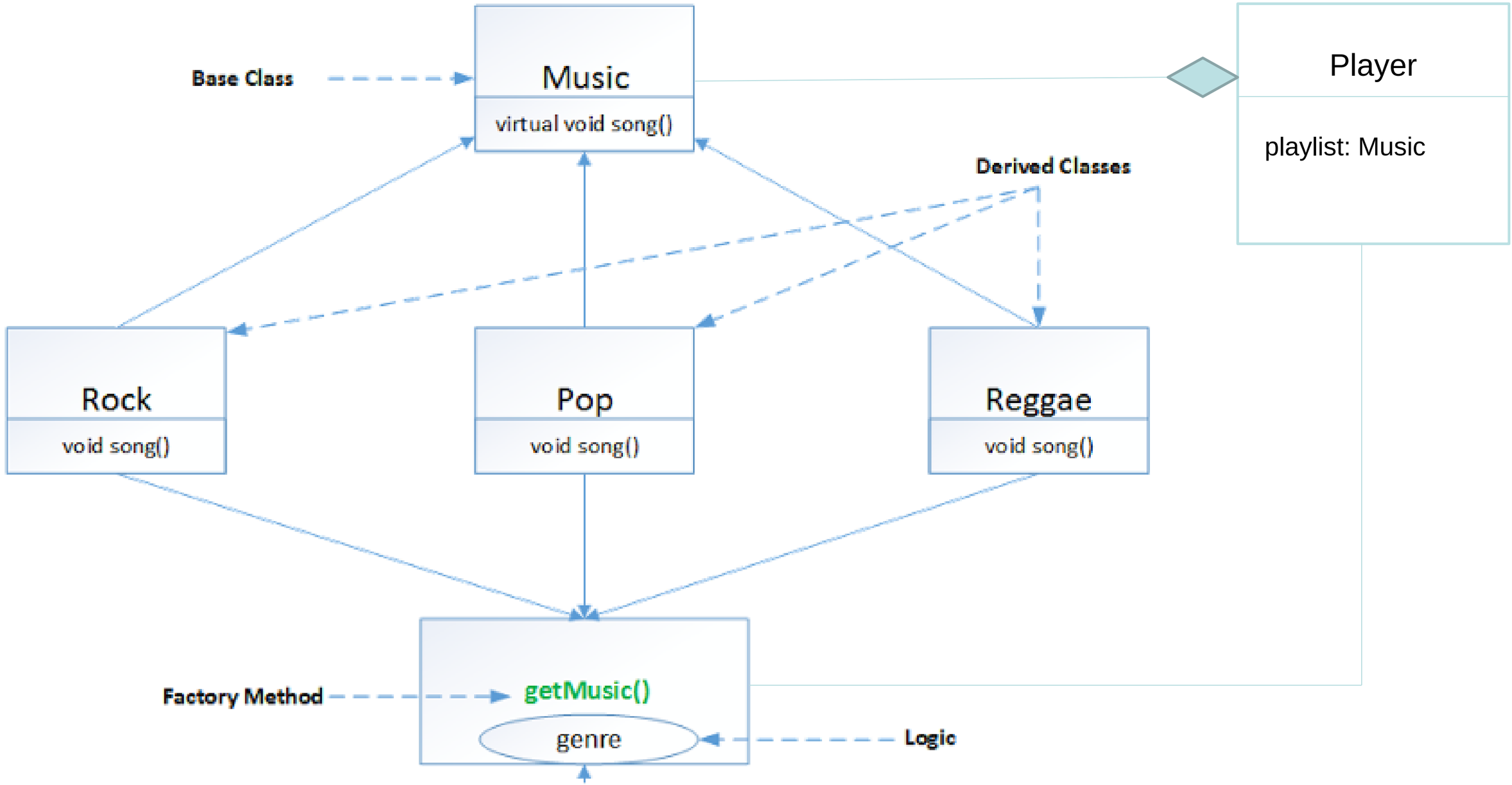
Gang of Four book
23 Design Patterns

Creational: Factory Method



Factory Method Pattern

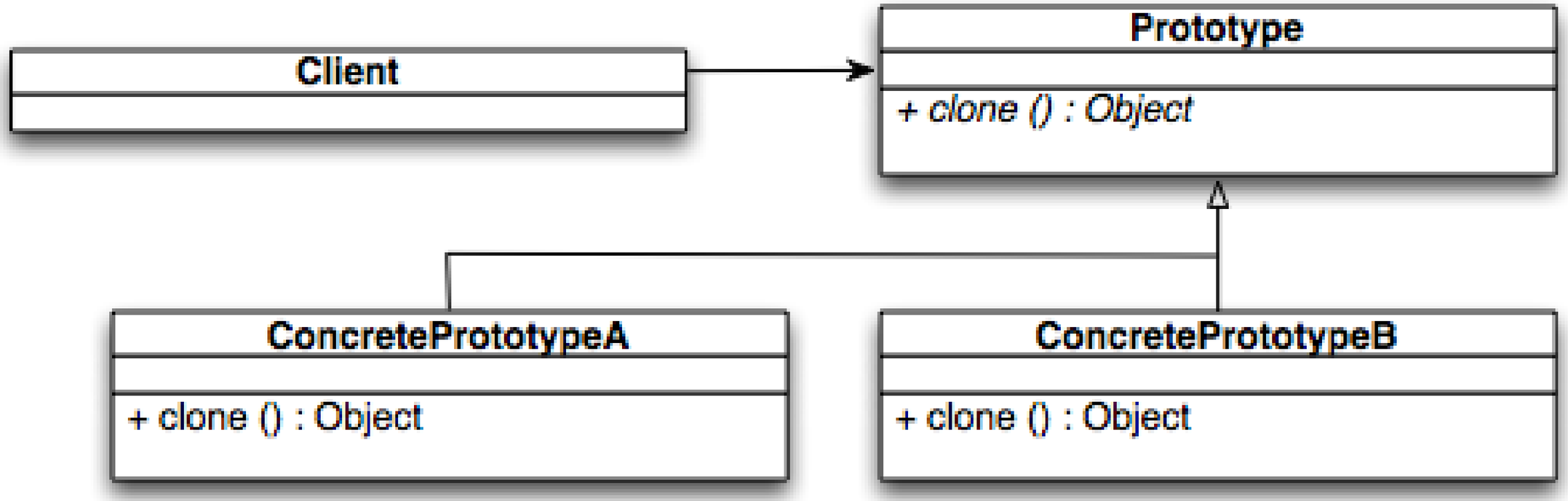
- Define an interface for creating an object, but let subclasses decide which class to instantiate.
- Various subclasses might implement the interface; the Factory Method instantiates the appropriate subclass based on information supplied by the client or extracted from the current state.


Base Class
Music
virtual void song()
Player
playlist: Music
Derived Classes
Rock
void song()
Pop
void song()
Reggae
void song()
Factory Method
getMusic()
genre
Logic

```
// Base Class
class Music{
  public:
    virtual void song() = 0;
};

// Derived class Rock from Music
class Rock:public Music{
  public:
    void song(){
      cout<<“Nirvana: Smells like a teen spirit”;
    }
};

// Derived class Pop from Music
class Pop:public Music{
  public:
    void song(){
      cout<<“Michael Jackson: Billie Jean”;
    }
};

// Derived class Rock from Music
class Reggae:public Music{
  public:
    void song(){
      cout<<“Bob Marley: No woman, No cry”;
    }
};
```

```
// Factory Class
class MusicFactory{
  public:
    //Factory Method
    Music * getMusic(string music){
      if(strcmp(music,”ROCK”)){
          return new Rock();
      }else if(strcmp(music,”POP”)){
          return new Pop();
      }else if(strcmp(music,”Reggae”)){
          return new Reggae();
      }else{
          return NULL;
      }
    }
};

// Main Program
int main(){
    // Create Factory
    MusicFactory * mf = new MusicFactory();
    // Factory instantiating an object of type ROCK
    Music * music = mf->getMusic(“ROCK”);
    if(music){
      music->song();
    }else{
      cout<<“Wrong Selection!”;
    }
}
```

Creational: Prototype



Prototype Pattern

- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- It speeds up the instantiation of very large, dynamically loaded classes (when copying objects is faster), and
- It keeps a record of identifiable parts of a large data structure that can be copied without knowing the subclass from which they were created.

Applicability and Examples

- Use Prototype Pattern when a system should be independent of how its products are created, composed, and represented, and:
  - Classes to be instantiated are specified at run-time
  - Avoiding the creation of a factory hierarchy is needed
  - It is more convenient to copy an existing instance than to create a new one.

Before Clone

```
class Stooge {
     public:
        virtual void slap_stick() = 0;
};

class Larry: public Stooge {
      public:
        void slap_stick() {
            cout << "Larry: poke eyes\n";
        }
};

class Moe: public Stooge {
      public:
        void slap_stick() {
            cout << "Moe: slap head\n";
        }
};
class Curly: public Stooge{
      public:
         void slap_stick() {
                  cout << "Curly: suffer abuse\n";
         }
};
```

```
int main() {
      vector <Roles *> roles;
      int choice;
      while (true) {
        cout << "Larry(1) Moe(2) Curly(3) Go(0): ";
        cin >> choice;
        if (choice == 0)
            break;
        else if (choice == 1)
            roles.push_back(new Larry);
        else if (choice == 2)
            roles.push_back(new Moe);
        else
            roles.push_back(new Curly);
      }

      for (int i = 0; i < roles.size(); i++)
            roles[i]->slap_stick();

      for (int i = 0; i < roles.size(); i++)
            delete roles[i];
}
```

After Clone

```
class Stooge {
public:
   virtual Stooge* clone() = 0;
   virtual void slap_stick() = 0;
};

class Factory {
public:
   static Stooge* make_stooge( int choice );
private:
   static Stooge* s_prototypes[4];
};

Stooge* Factory::s_prototypes[] = {
   0, new Larry, new Moe, new Curly };

Stooge* Factory::make_stooge( int choice ) {
   return s_prototypes[choice]->clone();
}

class Larry : public Stooge {
public:
   Stooge*   clone() { return new Larry(); }
   void slap_stick() {
      cout << "Larry: poke eyes\n"; }
};
```

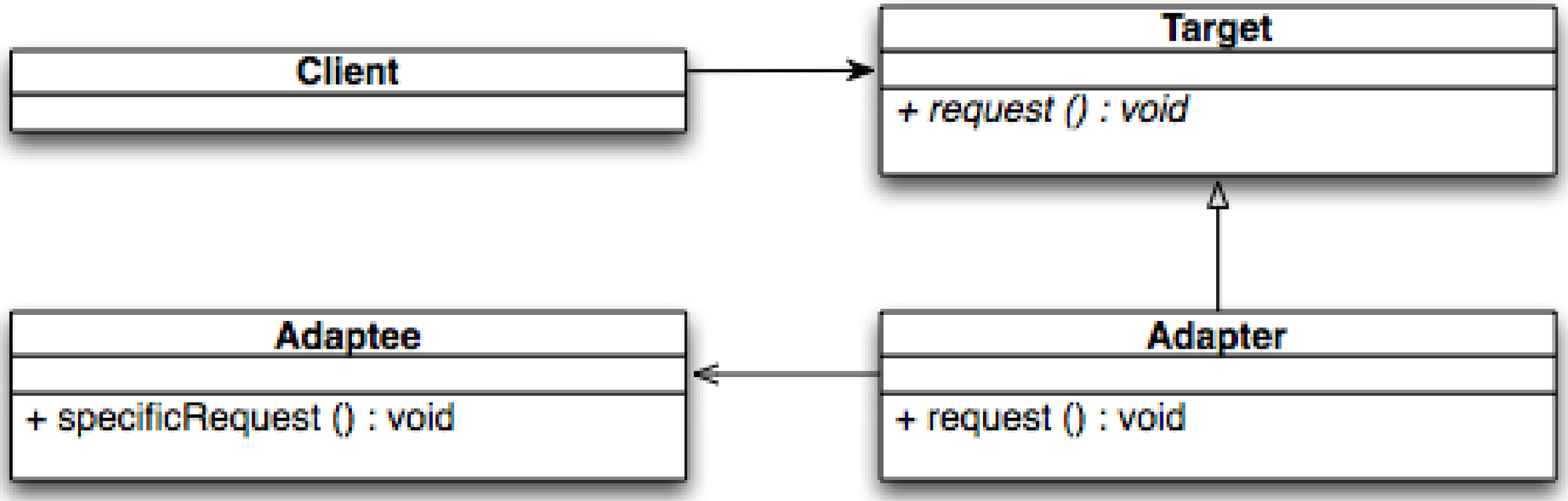
```
class Moe : public Stooge {
public:
   Stooge*   clone() { return new Moe(); }
   void slap_stick() {
      cout << "Moe: slap head\n"; }
};
class Curly : public Stooge {
public:
   Stooge*   clone() { return new Curly(); }
   void slap_stick() {
      cout << "Curly: suffer abuse\n"; }
};
int main() {
   vector <Roles *> roles;
   int choice;
   while (true) {
      cout << "Larry(1) Moe(2) Curly(3) Go(0): ";
      cin >> choice;
      if (choice == 0)
         break;
      roles.push_back(
         Factory::make_stooge( choice ) );
   }
   for (int i=0; i < roles.size(); i++)
      roles[i]->slap_stick();
   for (int i=0; i < roles.size(); i++)
      delete roles[i];
}
```

Example

In building stages for a game that uses a maze and different visual objects that the character encounters it is needed a quick method of generating the haze map using the same objects: wall, door, passage, room... The **Prototype pattern** is useful in this case because instead of hard coding (using new operation) the room, door, passage and wall objects that get instantiated, **CreateMaze** method will be parameterized by various prototypical room, door, wall and passage objects, so the composition of the map can be easily changed by replacing the prototypical objects with different ones.

The **Client** is the **CreateMaze** method and the **ConcretePrototype** classes will be the ones creating copies for different objects.

Structural: Adapter



Adapter Pattern

- Works as a bridge between two incompatible interfaces.
- It combines the capability of two independent interfaces.
- It involves a single class which is responsible to join functionalities of independent or incompatible interfaces.
- A real life example could be *a case of card reader which acts as an adapter between memory card and a laptop. You plug in the memory card into card reader and card reader into the laptop so that memory card can be read via laptop*.

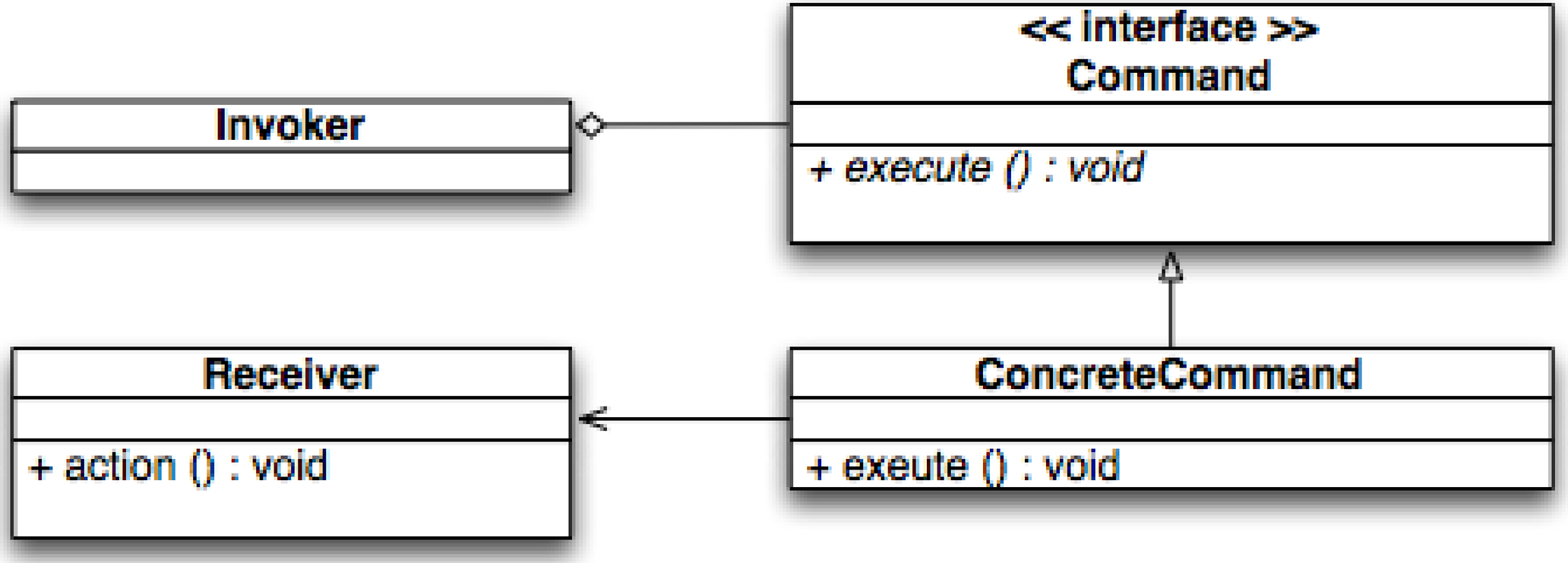
Example

```
class Circle{
  public:
    virtual void draw() = 0;
};

class StandardCircle{
  private:
    double _radius;
  public:
    StandardCircle(double radius){
       _radius = radius;
    }

    void oldDraw(){
       cout<<“StandardCircle: OldDraw”<<_radius;
    }
};
```

```
// Adapter Class
class CAdapter:public Circle, private StandardCircle{
  public:
    CAdapter(double diameter):
    StandardCircle(diameter/2){
       cout<<“Cadapter diameter:”<<diameter;
    }

    virtual void draw(){
       cout<<“CAdapter Draw”;
       oldDraw();
    }
};
```

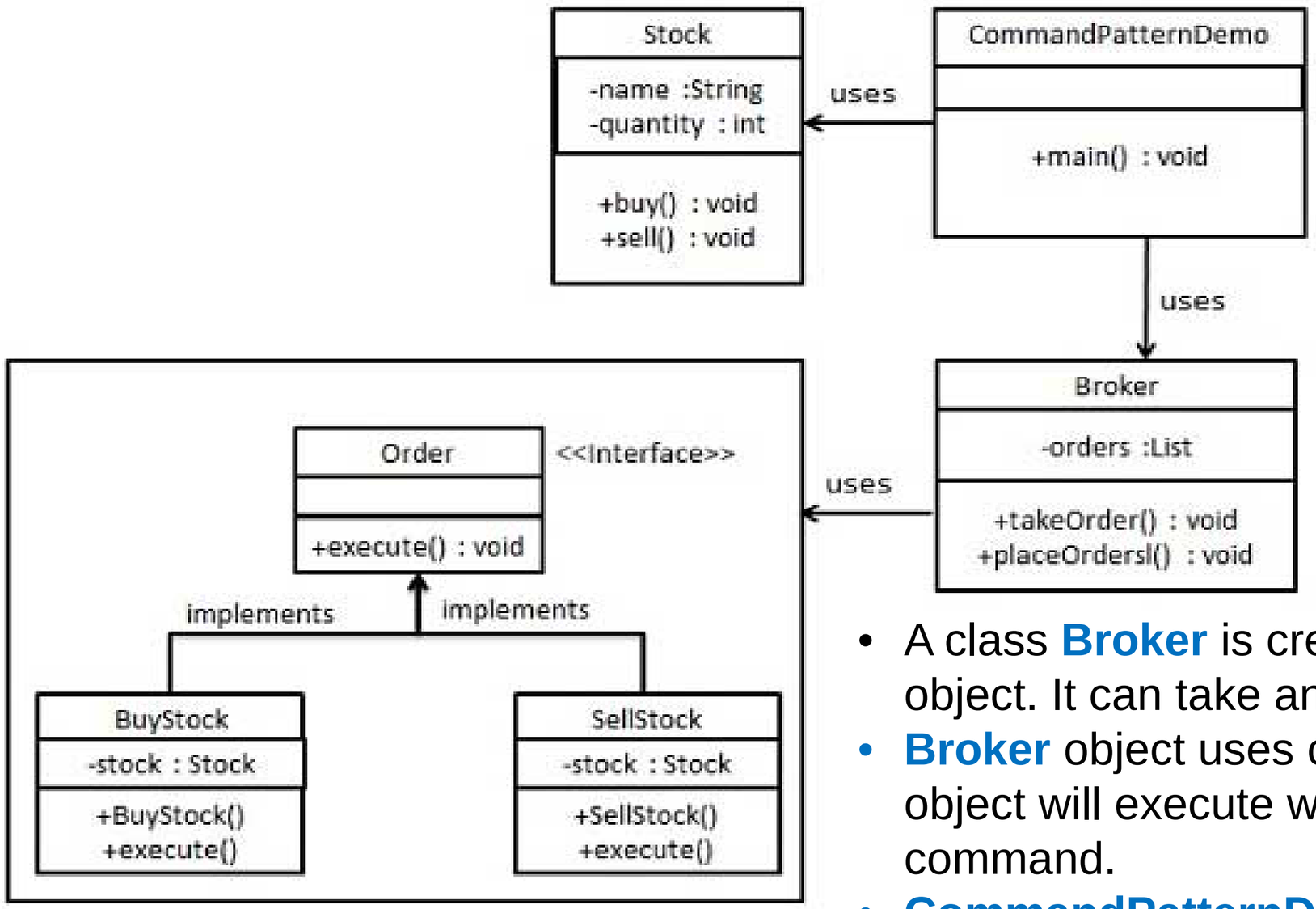
```
int main(){
    Circle * c = new CAdapter(14);
    c->draw();
    return 0;
}
```

Behavioural: Command



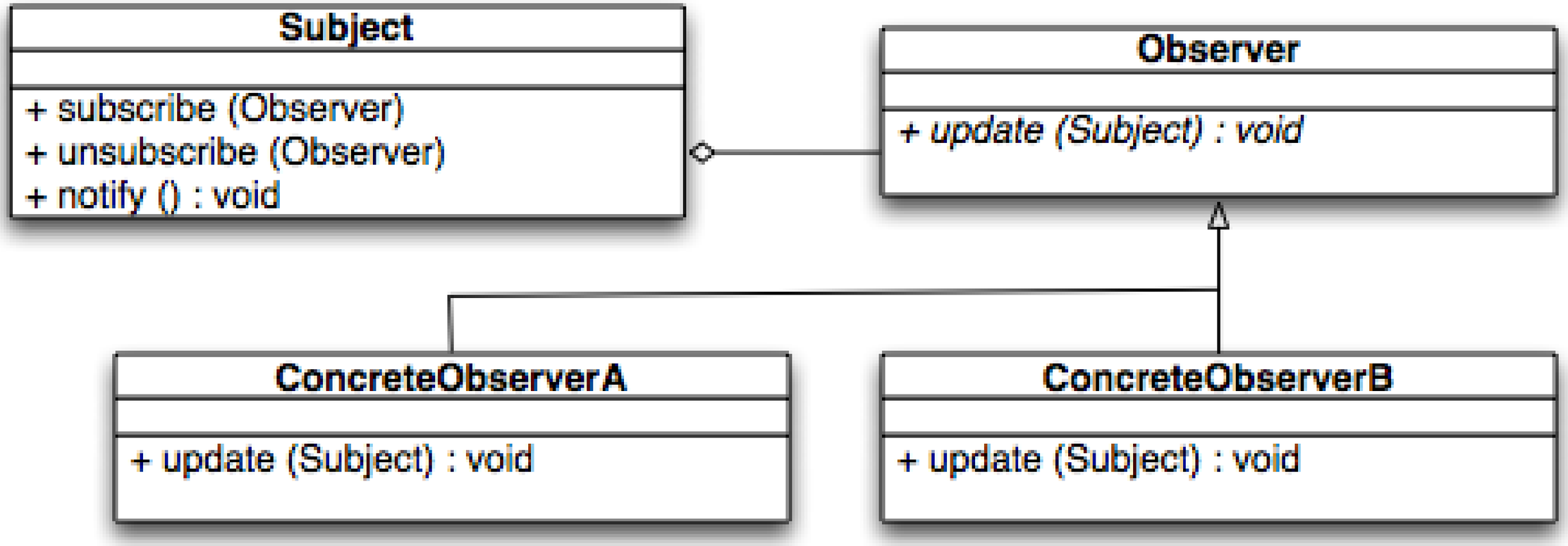
- *A request is wrapped under an object as command and passed to invoker object.*
- *Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.*

Example



- An interface **Order** which is acting as a command.
- A **Stock** class which acts as a request.
- We have concrete command classes **BuyStock** and **SellStock** implementing **Order** interface which will do actual command processing.
- A class **Broker** is created which acts as an invoker object. It can take and place orders.
- **Broker** object uses command pattern to identify which object will execute which command based on the type of command.
- **CommandPatternDemo**, our demo class, will use Broker class to demonstrate command pattern.

Behavioural: Observer

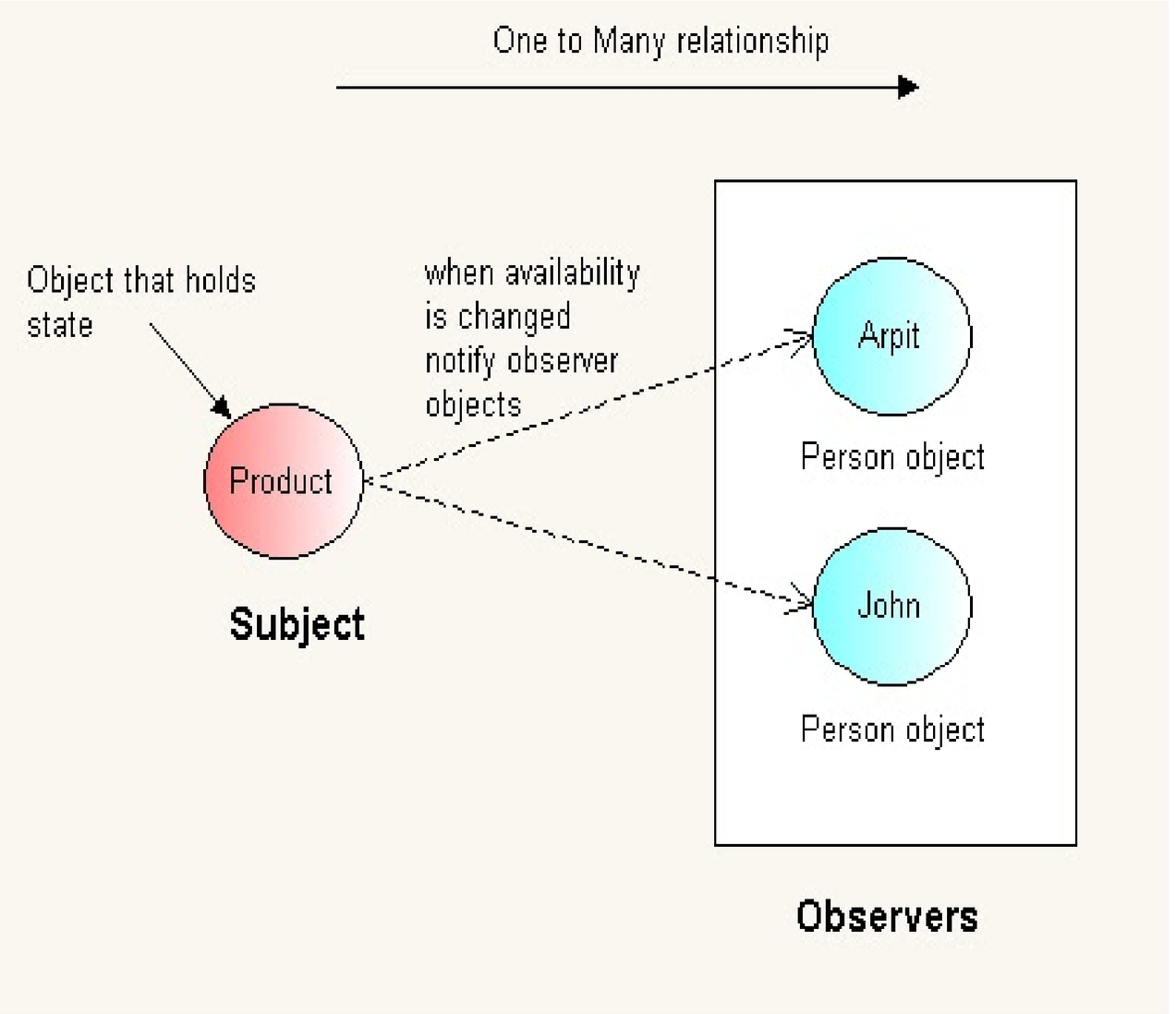


- Observer watch for any change in state or property of subject.
- Suppose you are interested in particular object and want to get notified when its state changes then you observe that object and when any state or property change happens to that object, it get notified to you.

Observer Pattern

Some real life examples:

- When you search for any product online and it is unavailable then there is option called “Notify me when product is available”. If you subscribe to that option then you will get notified when the product is available.
- On Facebook, if you subscribe someone then whenever new updates happen you will be notified.



Use Design Patterns to achieve and communicate good OO design

Design Patterns are based on two principles of reusable OO design

Program to an interface, not an implementation

Favour object composition over class inheritance

In short:
depend on abstractions

Implementations based on Design Patterns more readily achieve good design

Reusable

A reusable solution to a commonly occurring problem within a given context in software design.

Objects with recognized roles and responsibilities are more readily reused

Extensible

A design principle where the implementation takes future growth into consideration.

The ability to extend a system and the level of effort required to implement the extension.

Abstractions make code closed to modification and open to extension

Maintainable

The ease with which a software product can be maintained

Improved understanding through communication

Design Patterns provide a shared vocabulary of high-level concepts

Capture essential structural elements of an architecture

Codification of intuitive knowledge of experienced developers

Patterns are validated by experience

Communicate and justify key design decisions

Will you be able to recognize and use design patterns to solve problems?

Many design problems have already been solved.

Don’t reinvent the wheel

Commonly occurring Design Patterns have been identified and codified

Recognize Design Patterns
and use them to
convey and justify design decisions

Design Patterns lead to less work

This Week’s Tasks

Look Ahead: Plan remainder of semester

Supplementary Exercise: Conceptual Modelling

Supplementary Exercise: Case Study – Iteration 5