Polymorphism ด้วย C++

เทพพิทักษ์ การุญบุญญานันท์

เกริ่นนำ

- Inheritance ช่วยจำแนกออบเจกต์ต่าง ๆ ออกเป็นหมวดหมู่
  - ง่ายต่อการจัดการคุณสมบัติที่มีร่วมกัน
  - สร้างคลาสใหม่ได้ง่ายขึ้นโดยต่อเติมจากคลาสเก่า (ส่งเสริมการ reuse โค้ด)
  - สามารถ override คุณสมบัติบางประการของคลาสเก่าได้
- ที่ผ่านมาเป็น abstraction แบบมองจากคลาสใหม่ย้อนไปหาคลาสเก่าที่มีอยู่แล้ว
- Polymorphism จะเป็น abstraction ที่เตรียมการจากคลาสเก่าเพื่อรองรับคลาสใหม่ในอนาคต

กรณีตัวอย่าง

ต้องการพิมพ์ข้อมูลพนักงานประเภทต่าง ๆ

```
enum EmpType {
   STAFF,
   TMP_WORKER
};
```

```
class Employee {
   // ...
public:
   void print() const;

protected:
   Employee (EmpType type)
      : type (type) {}

protected:
   EmpType  type;
};
```

กรณีตัวอย่าง

```
class Staff : public Employee {
  // ...
public:
  Staff() : Employee (STAFF) {}
  void print() const;
};

class TempWorker : public Employee {
  // ...
public:
  TempWorker() : Employee (TMP_WORKER) {}
  void print() const;
};
```

กรณีตัวอย่าง

การพิมพ์ข้อมูลพนักงาน

```
void PrintEmployee (Employee* e)
{
  e->print();
  switch (e->type) {
    case STAFF:
      ((Staff*)e)->print();
      break;
    case TMP_WORKER:
      ((TempWorker*)e)->print();
      break;
  }
}
```

กรณีตัวอย่าง

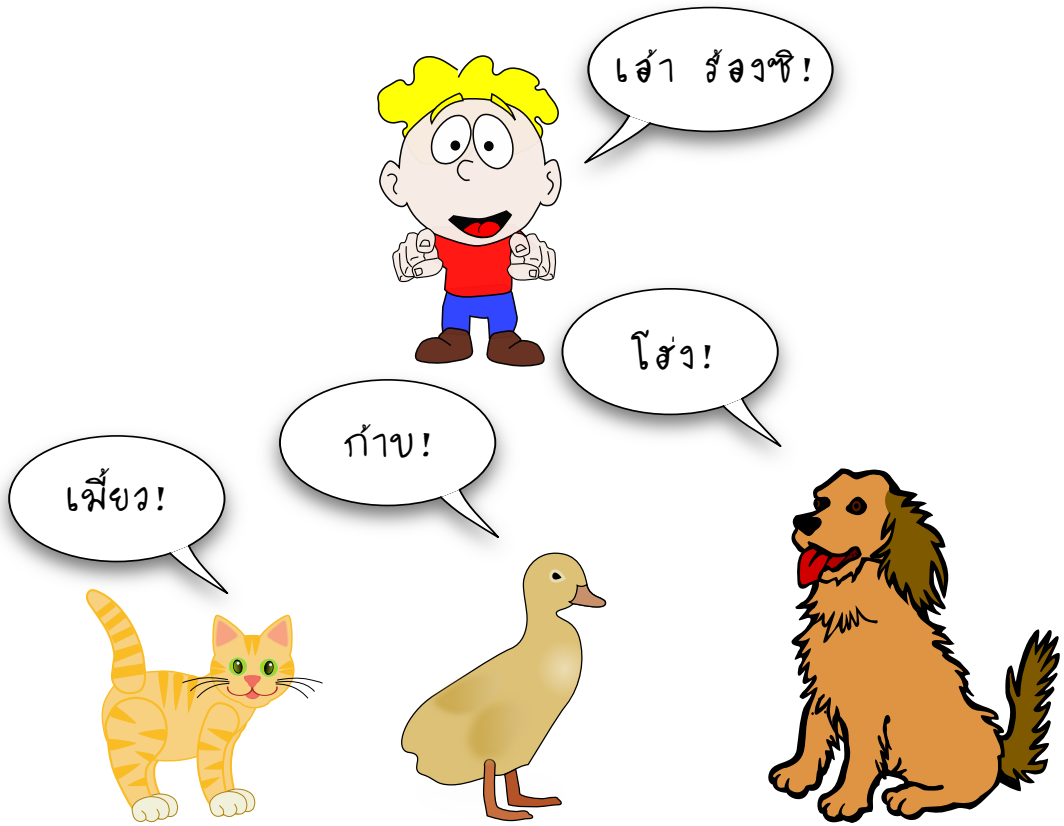
ข้อสังเกต

- ทุกฟังก์ชันในลักษณะเดียวกับ `PrintEmployee()` จะต้องมี common code และ switch-case เพื่อจัดการพนักงานทั้งสองชนิดแยกตามกรณี
- จะเกิดอะไรขึ้นถ้ามีพนักงานชนิดที่สาม (เช่น พนักงาน contract)?
- จะเกิดอะไรขึ้นถ้าต้องการแยกประเภทพนักงานประจำ (เช่น ระดับบริหาร, ระดับปฏิบัติการ)?

→ *วิธีนี้ไม่ scale!*

Polymorphism

การกำหนด operation ที่มีพฤติกรรมได้หลากหลายตามกรณีเฉพาะต่าง ๆ



Polymorphism

- รากศัพท์กรีก polys (many, much) + morphe (form, shape)
- กำหนด operation รูปทั่วไปใน base class
- กำหนดพฤติกรรมรูปแบบเฉพาะใน derived class
- ผู้เรียก เรียก operation ทั่วไปนั้นผ่าน base class
  - → พฤติกรรมเป็นไปตาม derived class ของออบเจกต์นั้น ๆ

Virtual Function ของ C++

- member function ที่สามารถถูก derived class override ได้ *แม้ขณะเรียกผ่าน base class*
- กำหนดโดยเพิ่ม keyword **`virtual`** หน้าการประกาศ member function

Virtual Function กับกรณีตัวอย่าง

```
class Employee {
  // ...
public:
  virtual void print() const;
};
```

Virtual Function กับกรณีตัวอย่าง

```
class Staff : public Employee {
  // ...
public:
  void print() const;
};

void Staff::print() const
{
  Employee::print();

  // print Staff info
  // ...
}
```

Virtual Function กับกรณีตัวอย่าง

```
class TempWorker : public Employee {
  // ...
public:
  void print() const;
};

void TempWorker::print() const
{
  Employee::print();

  // print TempWorker info
  // ...
}
```

Virtual Function กับกรณีตัวอย่าง

การพิมพ์ข้อมูลพนักงาน

```
void PrintEmployee (Employee* e)
{
  e->print();
}
```

Virtual Function กับกรณีตัวอย่าง

ข้อสังเกต

- ออบเจกต์ที่สร้างขึ้นจริงในโปรแกรมอาจเป็นชนิด **`Staff`** หรือ **`TempWorker`** แต่พอยน์เตอร์ไปยังออบเจกต์เหล่านี้สามารถส่งผ่านเป็นชนิด **`Employee*`** ได้
- การเรียก virtual function **`print()`** ผ่านพอยน์เตอร์ **`Employee*`** ซึ่งเป็น base class จะเป็นการเรียกฟังก์ชัน **`print()`** *ของ derived class ของออบเจกต์นั้น* แล้วแต่ว่าออบเจกต์นั้นจะเป็น **`Staff`** หรือ **`TempWorker`**
- เปรียบเทียบกับ non-virtual function การเรียกผ่านพอยน์เตอร์ **`Employee*`** จะเป็นการเรียกฟังก์ชันของ **`Employee`** เสมอ

Virtual Function กับกรณีตัวอย่าง

ข้อสังเกต

- จะเกิดอะไรขึ้นถ้ามีพนักงานชนิดที่สาม (เช่น พนักงาน contract)?
  - → สร้างคลาส **`ContractedEmployee`** โดย override virtual function **`print()`**
- จะเกิดอะไรขึ้นถ้าต้องการแยกประเภทพนักงานประจำ (เช่น ระดับบริหาร, ระดับปฏิบัติการ)?
  - → สร้างคลาส **`ExecutiveStaff`** และ **`OperationStaff`** โดย override virtual function **`print()`**

→ *ไม่ต้องแก้ switch-case ใด ๆ !*

Heterogeneous List

คลาส **Employee** ซึ่งมี polymorphism ทำให้สามารถเก็บ **Employee** ชนิดต่าง ๆ ใน heterogeneous list (ลิสต์ที่สมาชิกมีชนิดต่างกัน) ได้

```
class Personel {
  // ...
public:
  ~Personel();
  void printAll() const;

private:
  list<Employee*> employees;
};
```

Heterogeneous List

```
void Personel::printAll() const
{
  for (list<Employee*>::iterator i = employees.begin();
       i != employees.end(); ++i)
  {
    i->print();
  }
}
```

และเมื่อทำลาย **Personel** ก็ต้องทำลายออบเจกต์ต่าง ๆ ในลิสต์นี้ด้วย

```
Personel::~Personel()
{
  list<Employee*>::iterator i = employees.begin();
  while (i != employees.end()) {
    delete *i++;
  }
}
```

แต่การ **delete *i** จะเรียก destructor ของ ***i** ซึ่งเป็นชนิด **Employee** ไม่ใช่เรียก destructor ของคลาสจริง (**Staff** หรือ **TempWorker**)!

Virtual Destructor

พิจารณาคลาสทดสอบ:

```
class Base {
public:
  Base()  { cout << "Base c-tor" << endl; }
  ~Base() { cout << "Base d-tor" << endl; }
};

class Derived : public Base {
public:
  Derived()  { cout << "Derived c-tor" << endl; }
  ~Derived() { cout << "Derived d-tor" << endl; }
};
```

Virtual Destructor

```
int main()
{
  Base* p = new Derived;
  delete p;
  return 0;
}
```

ผลลัพธ์ของโปรแกรม:

```
Base c-tor
Derived c-tor
Base d-tor
```

กล่าวคือ destructor ของ derived class จะไม่ถูกเรียก

Virtual Destructor

แต่หากประกาศ destructor ของ base class ให้เป็น **virtual**:

```
class Base {
public:
  Base()  { cout << "Base c-tor" << endl; }
  virtual ~Base() { cout << "Base d-tor" << endl; }
};
```

ผลลัพธ์ของโปรแกรมจะกลายเป็น:

```
Base c-tor
Derived c-tor
Derived d-tor
Base d-tor
```

Virtual Destructor

- เมื่อ **`delete`** ผ่านพอยน์เตอร์ไปยัง base class ที่ประกาศ virtual destructor
  - → จะเรียก destructor โดยเริ่มจากคลาสจริงของออบเจกต์
- base class ที่มี virtual function จะมีโอกาสที่จะถูกบรรจุหรือเข้าถึงผ่านพอยน์เตอร์ที่ชี้ base class โดยที่คลาสจริงเป็น derived class คลาสใดคลาสหนึ่ง
  - → มีโอกาสที่จะถูก **`delete`** ผ่านพอยน์เตอร์ที่ชี้ base class
  - → ควรกำหนด virtual destructor

คำแนะนำ

คลาสที่มี virtual function ควรกำหนด virtual destructor ด้วยเสมอ

Virtual Destructor

- คลาส **Employee** ในตัวอย่างข้างต้นก็ควรประกาศ virtual destructor ด้วย

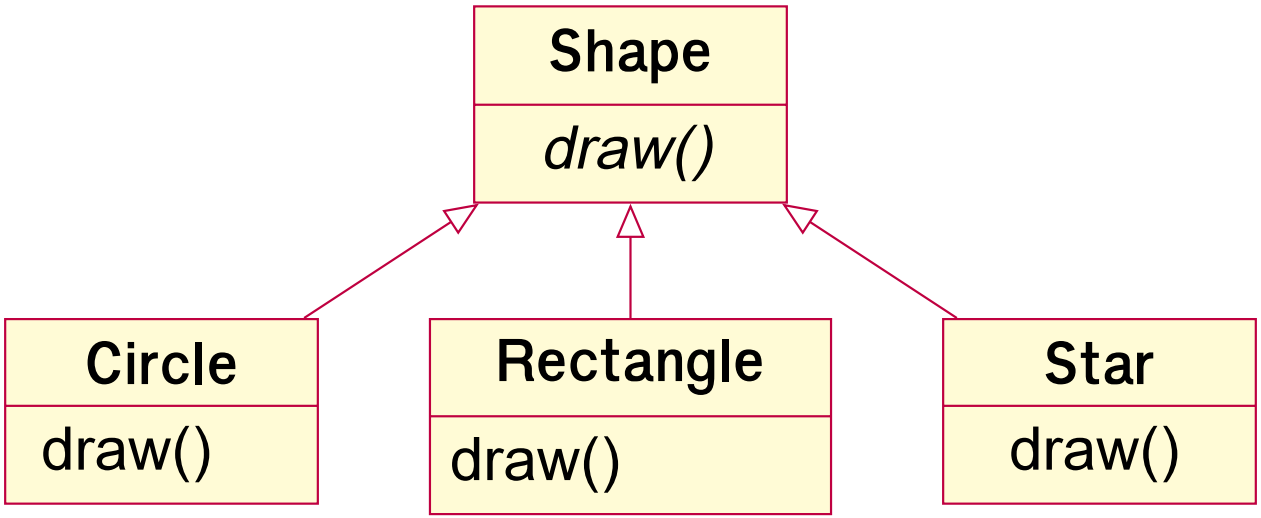
```
class Employee {
  // ...
public:
  virtual ~Employee() {}
  virtual void print() const;
};
```

หมายเหตุ:

- virtual destructor มี cost ในการเรียกเล็กน้อย จึงไม่ควรประกาศโดยไม่จำเป็น เช่น ในคลาสทั่วไปที่ไม่มี polymorphism

Abstract Class

virtual function ในบางกรณีก็ไม่มี default implementation ที่สมเหตุสมผล



Abstract Class

C++ อนุญาตให้กำหนด *“pure virtual function”* ได้ โดยต่อท้ายการประกาศ virtual function ด้วย initializer “= 0”

```
class Shape {
  // ...
public:
  virtual void draw() const = 0;
};
```

Abstract Class

- คลาสที่มี pure virtual function ถือว่าเป็น *“abstract class”*
- abstract class จะไม่สามารถมี instance เป็นออบเจกต์ได้

```
Shape s; // error! abstract class can't be instantiated!
```

- derived class ที่ implement pure virtual function ครบ จึงจะมี instance ได้ เรียกว่า *“concrete class”*
- derived class ที่ implement pure virtual function ไม่ครบ ก็ยังคงเป็น abstract class อยู่

Concrete Class

```
class Circle : public Shape {
public:
  Circle (double x0, double y0, double r)
    : center (x0, y0), radius (r) {}
  void draw() const; // implemented virtual function
private:
  Coord  center;
  double radius;
};

void Circle::draw() const
{
  // draw the circle
}
```

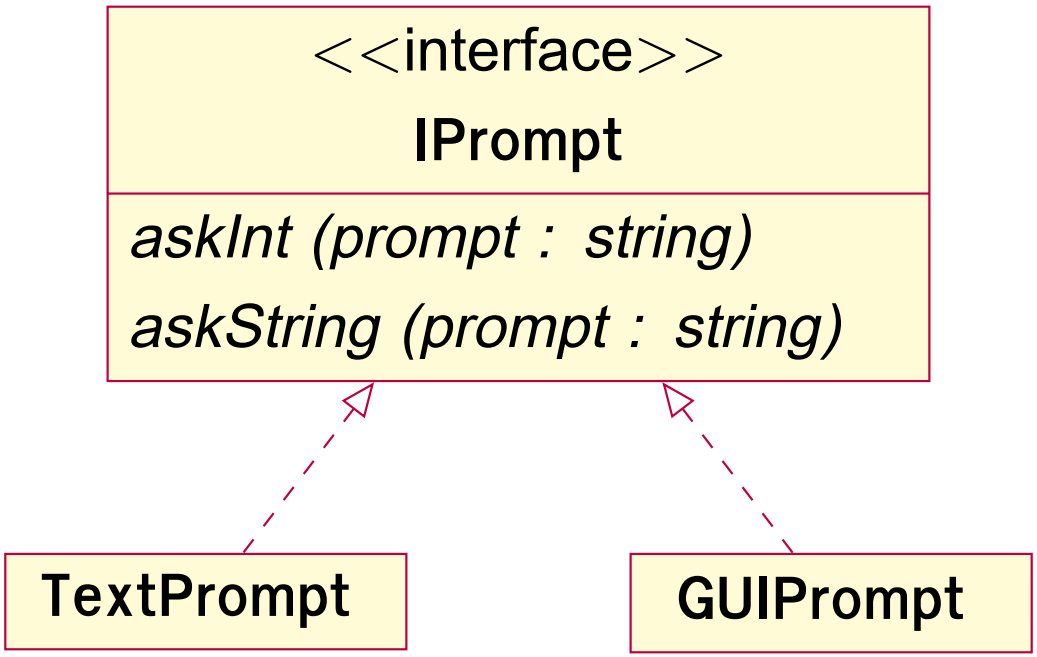
Interface

- *"interface"* ในทาง OOP คือช่องทางติดต่อสื่อสารระหว่างส่วนต่าง ๆ ของซอฟต์แวร์
- C++ แทน interface ได้ด้วย abstract class ที่มีสมาชิกเป็น pure virtual function ล้วน ๆ (ยกเว้น virtual destructor)

```
class IPrompt {
public:
  virtual ~IPrompt() {}
  virtual int    askInt (const char* prompt) = 0;
  virtual String askString (const char* prompt) = 0;
};
```

- ภาษาอื่นบางภาษา เช่น Java, C# มี keyword **`interface`** แยกต่างหากจาก **`class`**

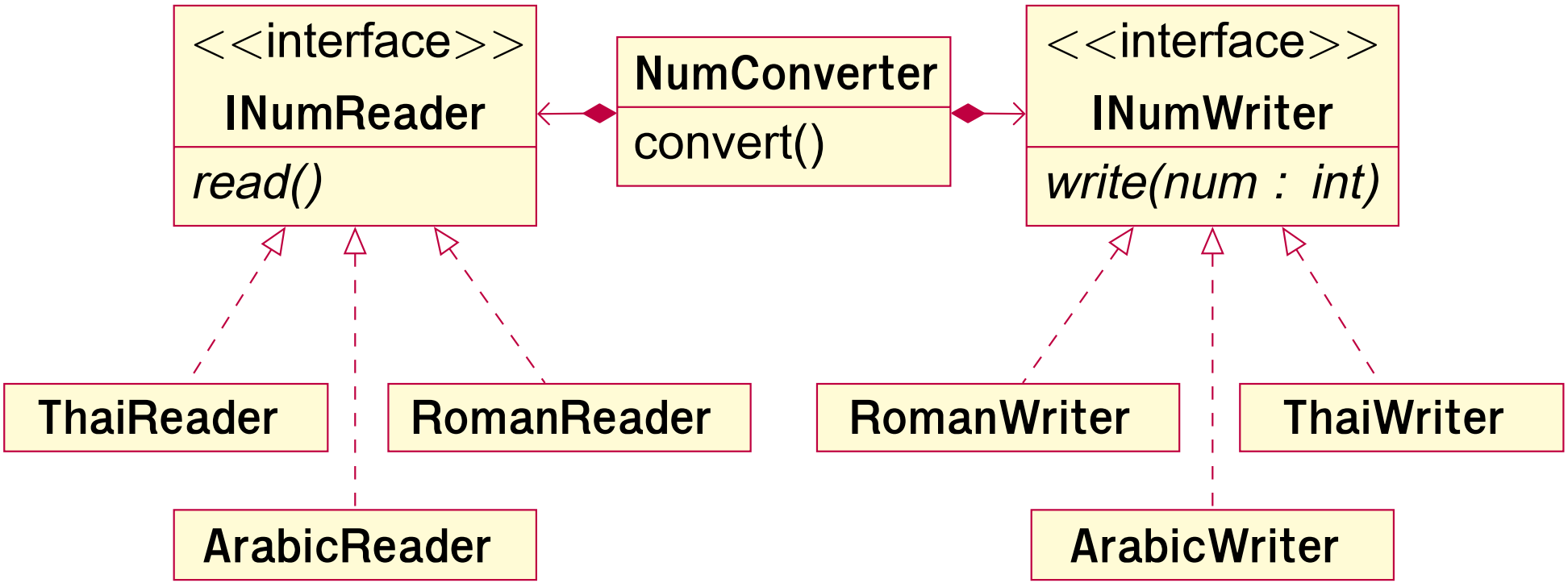
Interface ใน UML



- interface แทนด้วยคลาสที่มี stereotype «interface»
- การสร้างคลาสที่ derive จาก interface เรียกว่าการ *"realize"* interface แทนด้วย inheritance เส้นประ

ตัวอย่าง: โปรแกรมแปลงรูปแบบตัวเลข

โจทย์: เขียนโปรแกรมอ่านตัวเลข (เลขอารบิก, เลขไทย, เลขโรมัน ฯลฯ) และเขียนออกเป็นรูปแบบอื่นที่ต้องการ



ตัวอย่าง: โปรแกรมแปลงรูปแบบตัวเลข

- Interface

```
class INumReader {
public:
  virtual ~INumReader() {}
  // returns < 0 on input end
  virtual int read() = 0;
};

class INumWriter {
public:
  virtual ~INumWriter() {}
  virtual int write (int num) = 0;
};
```

ตัวอย่าง: โปรแกรมแปลงรูปแบบตัวเลข

- คลาส **NumConverter**

```
class NumConverter {
public:
  NumConverter (INumReader* reader, INumWriter* writer)
    : reader (reader), writer (writer) {}
  ~NumConverter() { delete reader; delete writer; }

  void convert();

private:
  INumReader* reader;
  INumWriter* writer;
};
```

ตัวอย่าง: โปรแกรมแปลงรูปแบบตัวเลข

- `NumConverter::convert()`

```
void NumConverter::convert()
{
  int num;
  while ((num = reader.read()) >= 0) {
    writer.write (num);
  }
}
```

ตัวอย่าง: โปรแกรมแปลงรูปแบบตัวเลข

- reader ทั้งหลาย

```
class ArabicReader : public INumReader {
public:
  int read() { /* read Arabic number */ }
};
class ThaiReader : public INumReader {
public:
  int read() { /* read Thai number */ }
};
class RomanReader : public INumReader {
public:
  int read() { /* read Roman number */ }
};
```

ตัวอย่าง: โปรแกรมแปลงรูปแบบตัวเลข

- writer ทั้งหลาย

```
class ArabicWriter : public INumWriter {
public:
  int write (int num) { /* write Arabic number */ }
};
class ThaiWriter : public INumWriter {
public:
  int write (int num) { /* write Thai number */ }
};
class RomanWriter : public INumWriter {
public:
  int write (int num) { /* write Roman number */ }
};
```

ตัวอย่าง: โปรแกรมแปลงรูปแบบตัวเลข

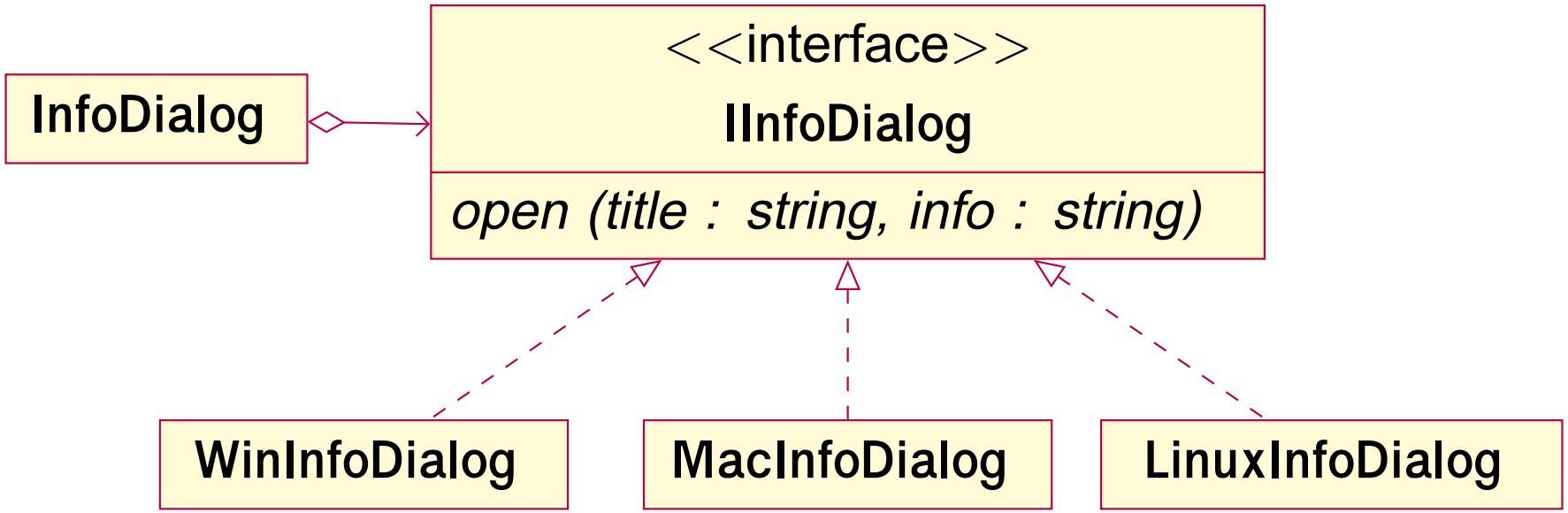
- ต้องการแปลงเลขอารบิกเป็นเลขโรมัน

```
int main()
{
  NumConverter c (new ArabicReader, new RomanWriter);
  c.convert();

  return 0;
}
```

การประยุกต์ใช้ Interface: Cross-Platform UI

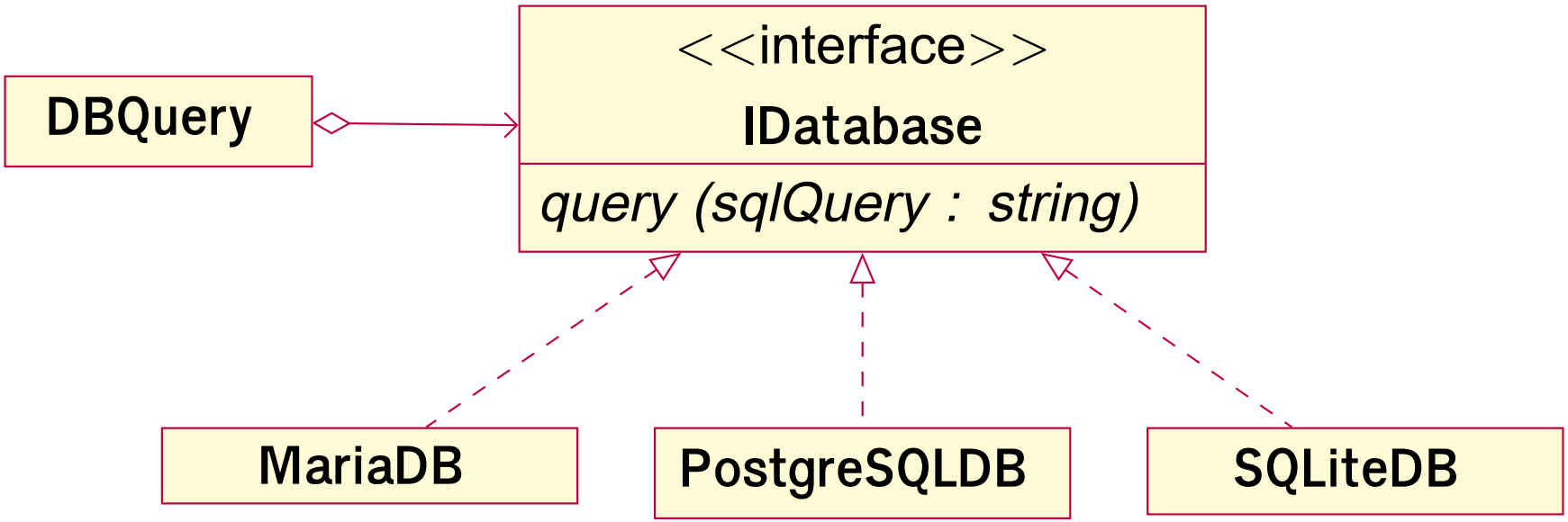
โปรแกรมที่ต้องการทำงานในหลายแพลตฟอร์ม (Windows, Mac, Linux, ...)



โค้ดแกนกลางมีชุดเดียว แต่เลือกคอมไพล์กับ implementation ต่าง ๆ แยกตามแพลตฟอร์ม

การประยุกต์ใช้ Interface: Pluggable Database Backends

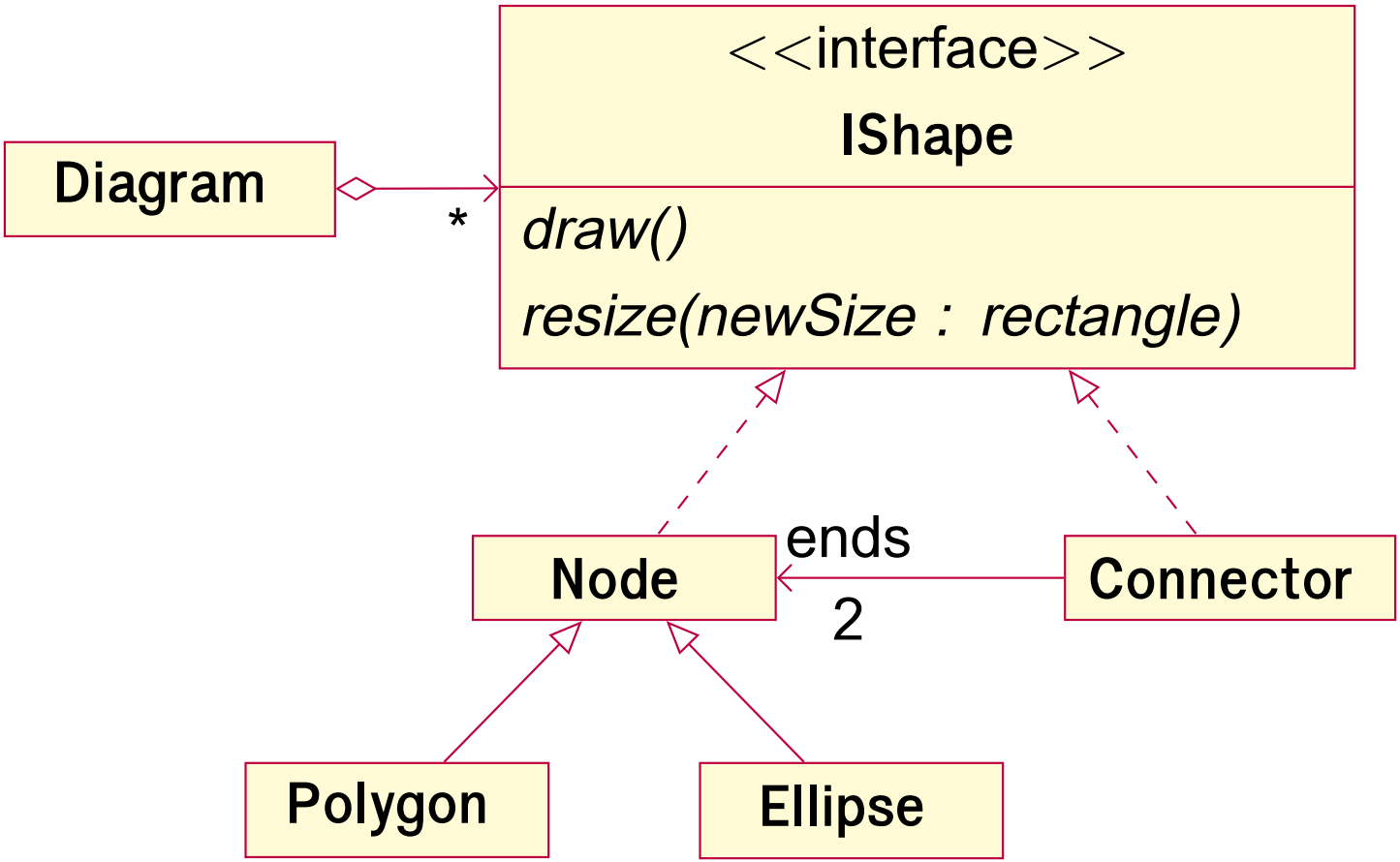
โปรแกรมที่ต้องการรองรับ DBMS ที่หลากหลาย (MariaDB, PostgreSQL, SQLite, . . .)



โค้ดแกนกลางมีชุดเดียว แต่เลือกโหลด plug-in ต่าง ๆ ตามความต้องการ

การประยุกต์ใช้ Interface: Diagram Editors

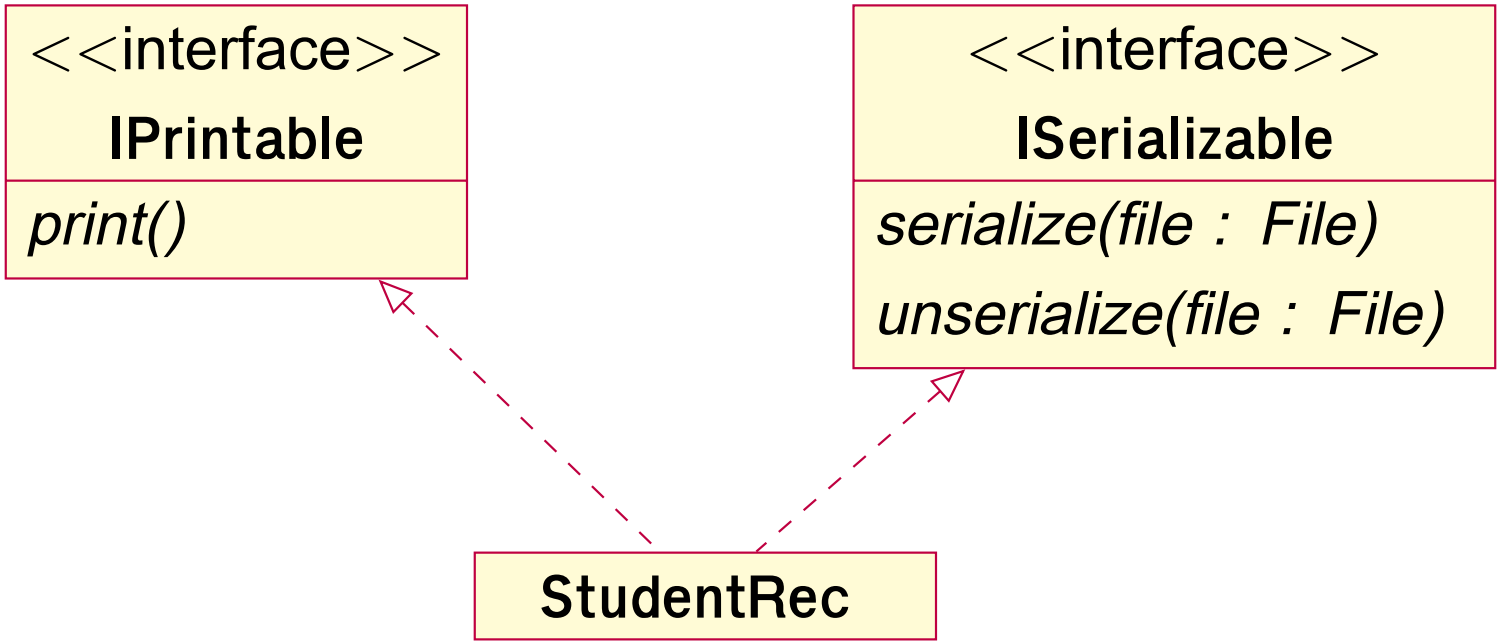
แผนภาพ (diagram) ประกอบด้วยภาพเรขาคณิตหลายรูปแบบผสมกัน



ออบเจกต์ชนิดต่าง ๆ ถูกสร้าง/ทำลายขณะทำงาน ตามที่ผู้ใช้สั่ง

Interface กับ Multiple Inheritance

คลาสหนึ่ง ๆ สามารถ realize interface ได้มากกว่า 1 interface



Interface กับ Multiple Inheritance

ใน C++ ทำได้ด้วย multiple inheritance

```
class IPrintable {
public:
  virtual ~IPrintable() {}
  virtual void print() const = 0;
};

class ISerializable {
public:
  virtual ~ISerializable() {}
  virtual bool serialize (File& file) = 0;
  virtual bool unserialize (File& file) = 0;
};
```

Interface กับ Multiple Inheritance

```
class StudentRec : public IPrintable,
                   public ISerializable
{
public:
  // IPrintable methods
  void print() const;

  // ISerializable methods
  bool serialize (File& file);
  bool unserialize (File& file);
};
```