Design Principles
Design Patterns
Good Practices

Programming Techniques II HS15

Rule of 5

```
class Foo {
  Foo(const Foo &) {  // copy constructor
    member = new double[999];
  }
  double * member;
};
```

Rule of 5

```
class Foo {
  Foo(const Foo &);          // copy constructor
  operator=(const Foo &);    // copy assignment operator
  Foo(Foo &&);               // move constructor
  operator=(Foo &&);         // move assignment operator
  ~Foo();                    // destructor
};
```

Either all of them, or none!

Auto-generation

	ctor	copy ctor	copy =	move ctor	move =	~dtor
ctor						
copy ctor						
copy =						
move ctor						
move =						
~dtor						

Auto-generation

	ctor	copy ctor	copy =	move ctor	move =	~dtor
ctor		✓	✓	✓	✓	✓
copy ctor	✓		✓	✗	✗	✓
copy =	✓	✓		✗	✗	✓
move ctor	✓	✗	✗		✗	✓
move =	✓	✗	✗	✗		✓
~dtor	✓	✓	✓	✗	✗	

Auto-generation

	ctor	copy ctor	copy =	move ctor	move =	~dtor
ctor		✓	✓	✓	✓	✓
copy ctor	✓		✓	✗	✗	✓
copy =	✓	✓		✗	✗	✓
move ctor	✓	✗	✗		✗	✓
move =	✓	✗	✗	✗		✓
~dtor	✓	✓	✓	✗	✗	

Rule of 5

```
class Abstract {
  virtual ~Abstract() { }   // allow polymorphism
};
```

Rule of 5

```
class Abstract {
  Abstract(const Abstract &) = default;
  Abstract & operator=(const Abstract &) & = default;
  Abstract(Abstract &&) = default;
  Abstract & operator=(Abstract &&) & = default;
  virtual ~Abstract() { }
};
```

Rule of 5 compromises

- copy ctor, copy =, ~dtor → non-movable
- move ctor, move =, ~dtor → non-copyable
- implicit move = deleted by:

  non-static const members, non-static & members,

  non-move-assignable members, non-move-assignable base

  implicit move ctor even more sensitive...
  more on that when we discuss move semantics!

Rule of zero

- Classes with custom rule-of-5 methods deal exclusively with ownership
- memory management follows *single responsibility principle*

  → smart pointers (next week)
- other classes should have no rule-of-5 methods

“Trust me...”

~ gcc

Good Design?

Fragile

Hard to reuse

Bad Design?

Hard to understand

Open/Closed Principle

```
class Rect {
    double get_width();
    double get_height();
};
```

```
double area(Rect r) {
    return r.get_width()
          *r.get_height();
}
```

Open/Closed Principle

```
class Rect: public Shape {
    double get_width();
    double get_height();
};
class Circle: public Shape {
    double get_radius();
};
```

```
double area(Shape s) {
  if(type(s)==type(Rect))
    return s.get_width()
           *s.get_height();
  if(type(s)==type(Circle))
    return s.get_radius()
           *s.get_radius();
}
```

Open/Closed Principle

```
class Rect: public Shape {
    double get_width();
    double get_height();
};
class Circle: public Shape {
    double get_radius();
};
class Triangle: public Shape {
    // ...
};
```

```
double area(Shape s) {
  if(type(s)==type(Rect))
    return s.get_width()
           *s.get_height();
  if(type(s)==type(Circle))
    return s.get_radius()
           *s.get_radius();
  if(type(s)==type(Triangle))
    // ...
}
```

Open/Closed Principle

- code should be *open for extension*, but *closed for modification*

```
class Shape {
    virtual double get_area() = 0;
};
double area(Shape s) {
    return s.get_area();
}
```

Single Responsibility

```
class Animal {
  std::string get_name() const;
  void run();
  std::string mate_with(Animal &);
  void save_to_file(std::string &) const {
    // open file, write data, close file
  }
};
```

- who uses functionality from `Animal`?

  → a population class, another animal, a save feature, …

Single Responsibility

```
class Animal {
  // ...
  properties & return_data();
};
void save(std::ostream & o,
          Animal & a) {
  for(auto p: a.properties) {
    o << p;
  }
}
```

```
void saveHTML(std::ostream
          & o, Animal & a) {
  for(auto p: a.properties) {
    o << "<" << p.tag << ">"
      << p.content
      << "</" << p.tag << ">";
  }
}
```

Single Responsibility

- leads to lower coupled design / dependencies
- but... can lead to getter / setter / friend bloat

- how easy is it to give your class a name?
- can you identify subtasks to refactor?

Hint: PennaLV genome

Interface Segregation

```
class Animal {
  virtual void make_child()  = 0;
  // ...
  virtual wool_t give_wool() = 0;
  virtual void eat_grass() = 0;
};
class Sheep : public Animal {
  // …
  wool_t give_wool() override {
      // happy shearing
  }
};
```

```
class Bear : public Animal {
  // ...
  void eat_grass() override {

  }
}
```

Don't force dependencies on unused interfaces!

Formal Principles

- SOLID (single responsibility, open/closed, Liskov substitution, interface segregation, dependency inversion)
- DRY (don't repeat yourself) ↔ WET (we like typing)
- SSOT, KISS, ...

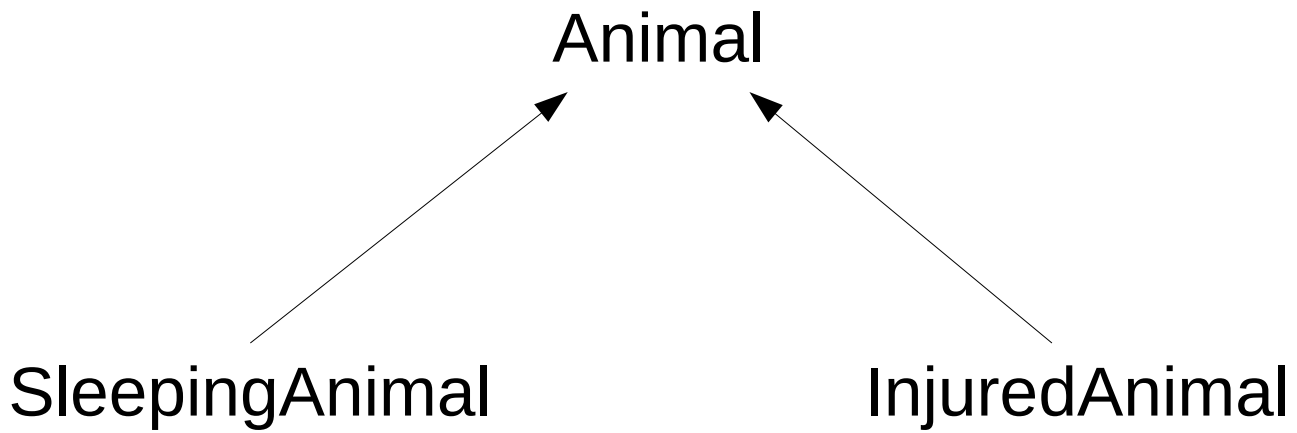
Factory Pattern

```
class AnimalFactory {
    static Animal * make_child(int choice) {
        switch(choice) {
            Case 0: return new Sheep();
            Case 1: return new Bear();
        }
    }
};
```

! use smart pointers !

Useful to provide instances to a container or unit test

Decorator



Adjust existing functionality, interchangeable with base

Decorator

```
class Animal {
  virtual void eat() {
    // with appetite
  };
};
class SleepingInjuredAnimal
  : public InjuredAnimal
  , public SleepingAnimal {
  virtual void eat() {
    SleepingAnimal::wake_up();
    InjuredAnimal::eat();
  }
};
```

```
class InjuredAnimal : public Animal
{
  virtual void eat() {
    moan();
    Animal::eat();
  }
};
class SleepingAnimal : public
Animal {
  virtual void eat() {
    wake_up();
    Animal::eat();
  }
};
```

Composition (holding the wrapped class in a member) scales better!

Recursively Bounded Quantification

Curiously Recurring Template Pattern (CRTP)

Curiously Recurring Template Pattern (CRTP)

```
template <class D>
class Base {
  void foo() {
    std::cout << D::x << std::endl;
  }
};

class Derived : public Base<Derived> {
    int x;
};
```

Curiously Recurring Template Pattern (CRTP)

```
template <class D>
class Base {
  void foo() {
    static_cast<const D*>(this)->foo();
  }
};

class Derived : public Base<Derived> {
    void foo() const {}
};
```

Curiously Recurring Template Pattern (CRTP)

```
template <class D>
class Base {
    void foo() { /* do something with D members */ }
    int x;
};
class Derived : public Base<Derived> {
};
int main() {
    Derived d;
    d.foo();
    std::cout << d.x << std::endl;
    return 0;
}
```