

# Factories

Computer Science and Engineering ■ College of Engineering ■ The Ohio State University

## Lecture 20

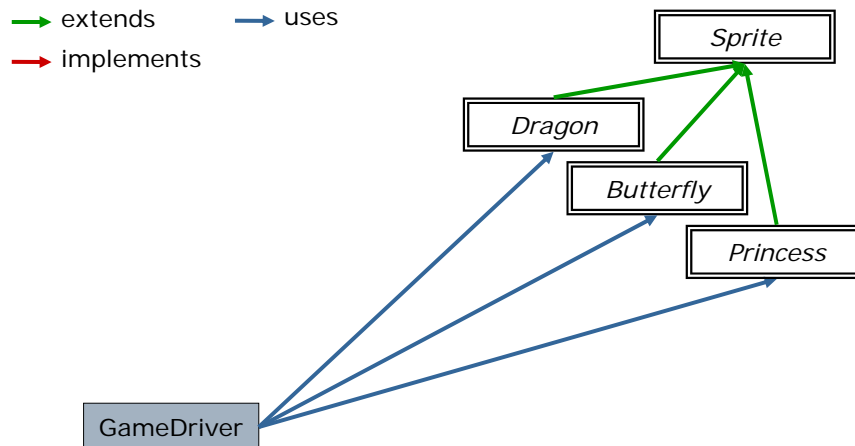
# A Game of Sprites

Computer Science and Engineering ■ The Ohio State University

- Consider a game consisting of sprites
  - Dragons, butterflies, princesses
- Main class: GameDriver
  - Populates the world with sprites
  - Responds to user events (eg mouse clicks)
  - Draws, erases, and moves sprites
  - Keeps track of score
- GameDriver is coded to the interface
  - Sprite interface promises generic drawing and moving abilities
  - Specific kinds of sprites have more behaviors (eg breathing fire)

# Sprites Hierarchy

Computer Science and Engineering ■ The Ohio State University



# Instantiating Objects

Computer Science and Engineering ■ The Ohio State University

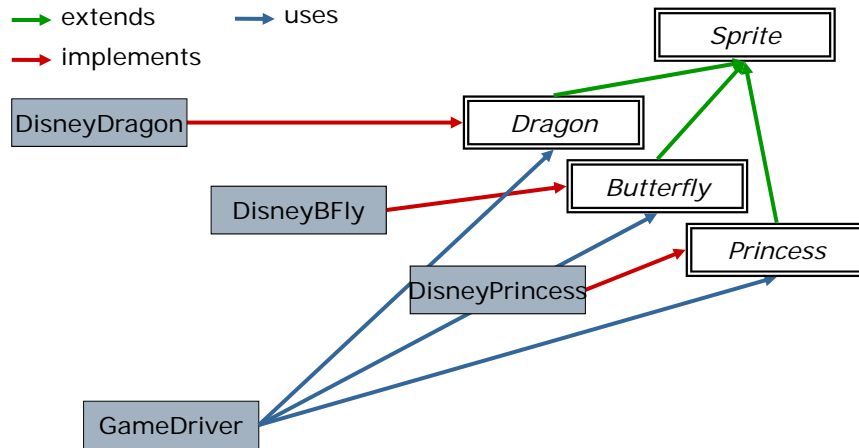
- GameDriver is general (coded to the interface)

```
class GameDriver {
    private List<Dragon> dragons;
    private List<Butterfly> butterflies;
    . . .
    public boolean isQueen(Princess p) {...}
}
```

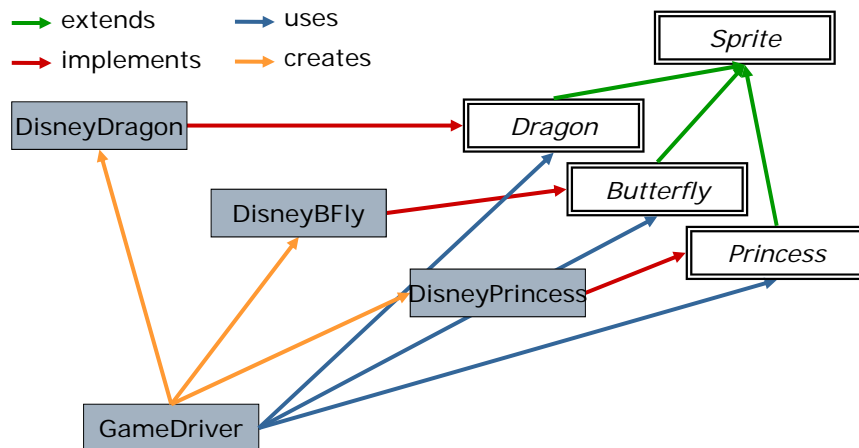
- But every call to new requires a *class*

```
public void populate() {
    Dragon villain = new DisneyDragon(35);
    . . .
}
```

# Sprites Hierarchy



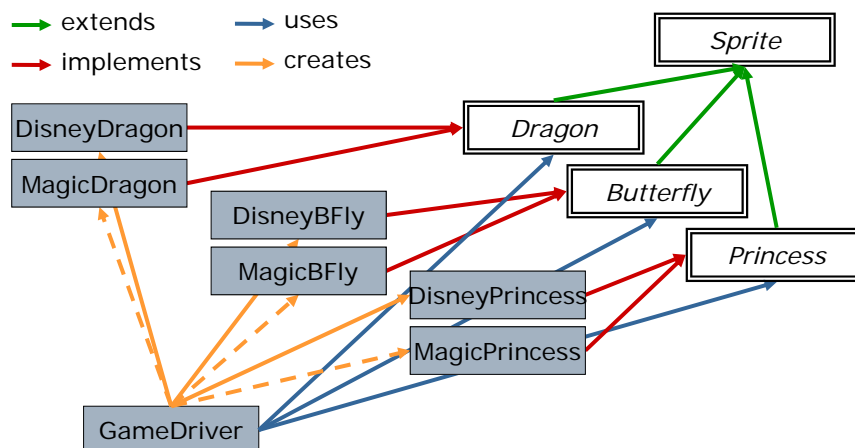
# Sprites Hierarchy



# Product Lines

- Object creation may occur in many different places
  - Across the program, every method that creates a sprite
  - Across a method, every line that creates a sprite
- Some classes may be designed to work best with other classes
  - An example of concrete-concrete coupling (generally a bad thing)
  - Example: themes for our game of sprites
    - Disney characters vs Magic characters
- Goal: Single-point-of-control over which product line is used
  - Every instantiation should be a Disney character
  - Should be easy to switch to all Magic characters

# Sprites Hierarchy



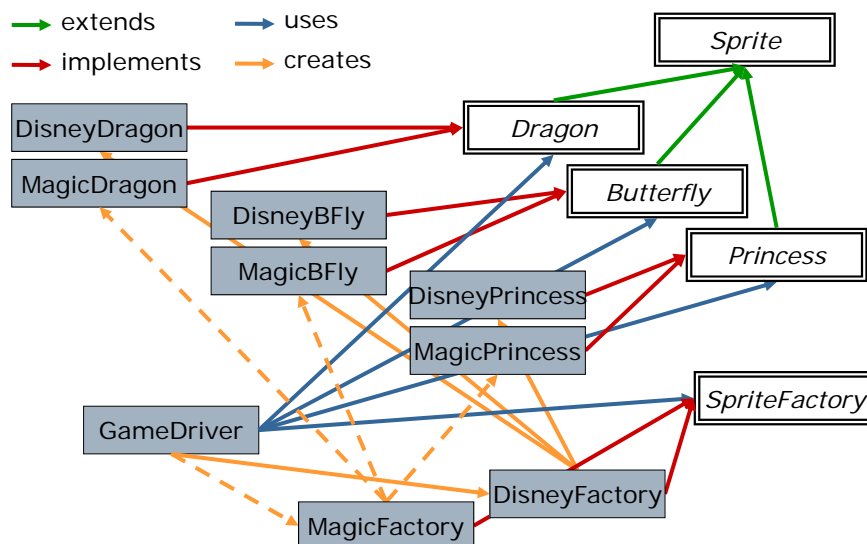
# Solution: Factory Component

- Add a level of indirection
- Responsibility for instantiation of sprites encapsulated in one place: a factory
  - Factory object can create Dragon, Butterfly, and Princess objects

```
interface SpriteFactory {  
    Dragon createDragon(int size);  
    Butterfly createButterfly();  
    Princess createPrincess(String name);  
}
```
  - Each implementation of Factory creates a single product line

```
class MagicFactory implements SpriteFactory {  
    public Dragon createDragon(int size) {  
        return new MagicDragon(size);  
    }  
    ...  
}
```
- Known as the “Factory Pattern” (a creational design pattern)

# Sprites Hierarchy with Factory



# Instantiating Objects: Factory

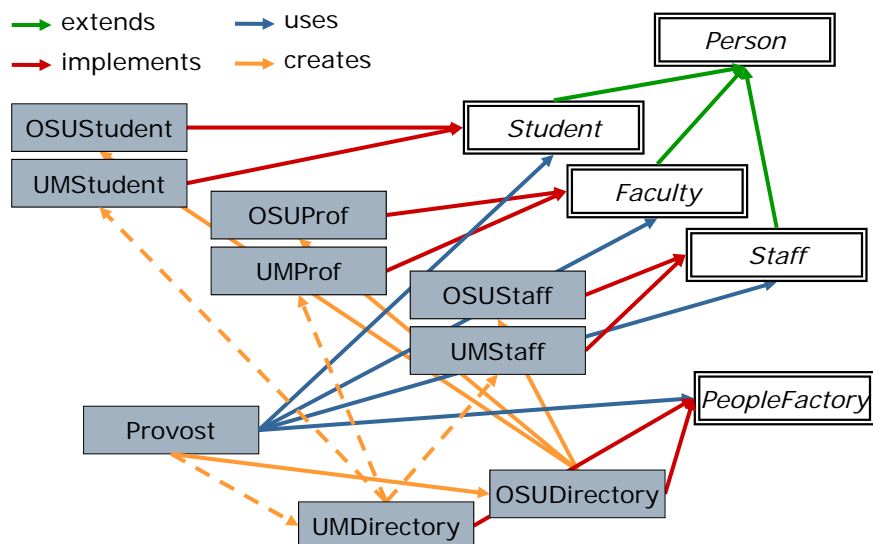
- GameDriver now creates *factory*:

```
class GameDriver {  
    private List<Dragon> dragons;  
    private List<Butterfly> butterflies;  
    . . .  
    private SpriteFactory factory =  
        new DisneyFactory();  
}
```

- Methods use *factory* to create objects:

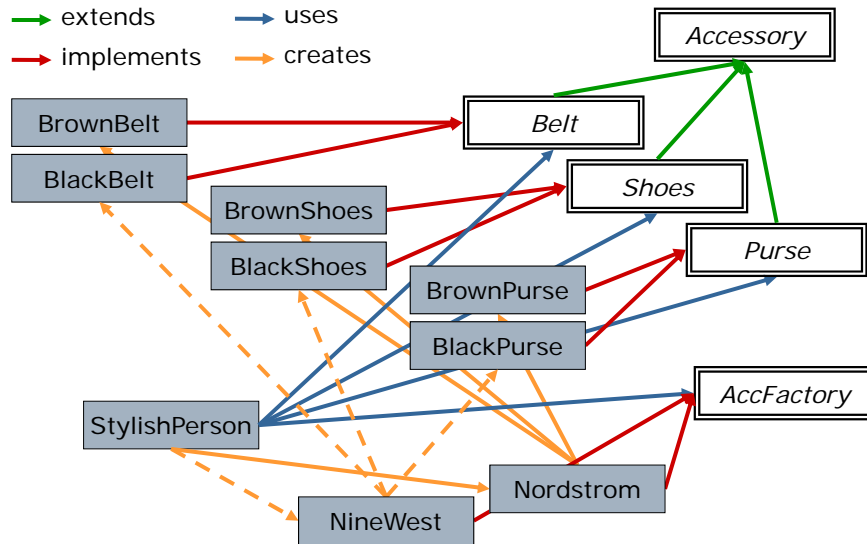
```
public void populate() {  
    Dragon villain = factory.createDragon(10);  
    . . .  
}
```

# Person Hierarchy with Factory



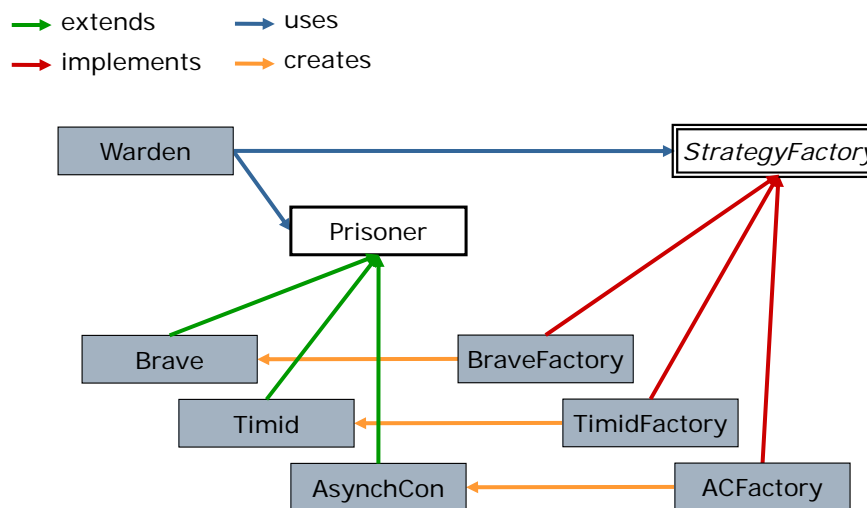
# Accessory Hierarchy with Factory

Computer Science and Engineering ■ The Ohio State University



# Warden and Prisoners

Computer Science and Engineering ■ The Ohio State University



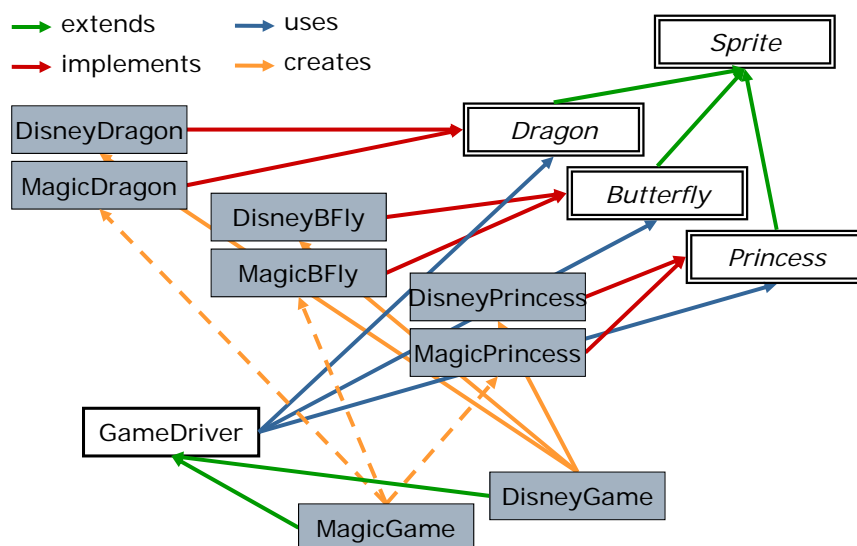
## Alternative: Factory Method

Computer Science and Engineering ■ The Ohio State University

- A different creational pattern
- Instantiation encapsulated in *method*
  - Class can have larger responsibilities
- This method designed to be overridden
  - Subclasses differ in the product line from which the overridden method creates new instances
- Distinction between these two patterns:
  - In abstract factory pattern, the factory class is responsible *only* for creation
  - In factory method pattern, the class containing the factory method is responsible for *both* creation and use/assembly

## GameDriver with Factory Method

Computer Science and Engineering ■ The Ohio State University



## Recall Basic JUnit Recipe

Computer Science and Engineering ■ The Ohio State University

- Given class SmartPerson implements interface Person
- Separate fixture into:
  - Base class testing behavior promised in Person
  - Derived class testing implementation-specific behavior of SmartPerson
- Base class contains:
  - Protected member of (declared) type Person
  - Abstract @Before method to initialize this member
- Derived class contains:
  - Overridden version of @Before to instantiate a SmartPerson

## JUnit with Inheritance

Computer Science and Engineering ■ The Ohio State University

→ extends  
→ implements



PersonTest

SmartPersonTest

```
protected Person p;  
@Before  
public abstract void setUp();  
@Test  
public void someTest1() {...}  
@Test  
public void someTest2() {...}
```

```
@Override @Before  
public void setUp() {  
    p = new SmartPerson();  
}
```

## Base Class Test Fixture

Computer Science and Engineering ■ The Ohio State University

```
class PersonTest {
    protected Person p1;
    protected Person p2;
    @Before
    public abstract void setUp();

    @Test
    public void doesSum() {
        int actual = p1.add(3,4);
        int expected = 7;
        assertTrue((actual - expected <= 2)
            && (actual - expected >= -2));
    }
}
```

## Derived Class Test Fixture

Computer Science and Engineering ■ The Ohio State University

```
class SmartPersonTest extends PersonTest {
    @Override
    @Before
    public void setUp() {
        p1 = new SmartPerson();
        p2 = new SmartPerson("Evariste Galois");
    }

    @Test
    public void doesSumAccurately() {
        assertEquals(7, p1.add(3,4));
    }
}
```

## JUnit with Factory Methods

Computer Science and Engineering ■ The Ohio State University

- Current recipe resembles a factory method
  - @Before method overridden and responsible for instantiation
- Limitation: JUnit fixture methods (like setup) can not have arguments
  - Derived class instantiates the members
  - Constructor arguments are fixed in body of setup
- Goal: Permit test cases to construct their *own* instances for testing
  - Desirable when there are many boundary conditions not easily covered by a small number of statically-instantiated objects

## New Base Class Test Fixture

Computer Science and Engineering ■ The Ohio State University

```
class PersonTest {
    protected Person p;

    protected abstract Person
        createFromString(String name);

    @Test
    public void doesSum() {
        p = createFromString("Galileo Galilei");
        int actual = p.add(3,4);
        int expected = 7;
        assertTrue((actual - expected <= 2)
            && (actual - expected >= -2));
    }
}
```

## New Derived Class Test Fixture

Computer Science and Engineering • The Ohio State University

```
class SmartPersonTest extends PersonTest {
    @Override
    protected Person
        createFromString(String name) {
            return new SmartPerson(name);
        }

    @Test
    public void doesSumAccurately() {
        p = createFromString("Galileo Galilei");
        assertEquals(7, p1.add(3,4));
    }
}
```

## Good Practice: Static Factories

Computer Science and Engineering • The Ohio State University

- ❑ Class provides a public *static* factory method
  - Return type is an instance of the class
  - `public static Integer valueOf(int i);`
- ❑ Advantages:
  - Factories can have descriptive names
  - `BigInteger p = BigInteger.probablePrime(128, rnd);`
  - Need not create a *new* instance!
    - ❑ For immutables, return reference to *existing* instance
    - ❑ For example, which is better?  
`Integer i1 = new Integer(1);`  
`Integer i2 = Integer.valueOf(1);`
  - Advanced technique: return instance of a private class
    - ❑ Client knows nothing about class, only the interface
- ❑ Disadvantages:
  - No public/protected constructor means no subclassing
  - No real distinction from any other static method
- ❑ Naming conventions: `valueOf()`, `getInstance()`

## Summary

Computer Science and Engineering • The Ohio State University

- Creation with `new()` gives concrete-to-concrete coupling
  - Product lines difficult to enforce/support
- Abstract factory pattern
  - Creation delegated to special-purpose class
  - Factory class designed to be extended
  - Each subclass creates objects from one product line
- Factory method pattern
  - Specific creational methods designed to be overridden
  - Each subclass overrides method to create objects from one product line
- Implications for JUnit
- Static factory methods