A Game of Sprites

- Consider a game consisting of sprites
  - Dragons, butterflies, princesses
- Main class: GameDriver
  - Populates the world with sprites
  - Responds to user events (e.g., 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 (e.g., breathing fire)
Sprites Hierarchy

```
extends uses
`Implementes
```

Instantiating Objects

- GameDriver is general (coded to the interface)

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

- But every call to new requires a `class`

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

- extends
- implements
- uses
- creates

Sprite
  - DisneyDragon
  - Dragon
  - Butterfly
  - Princess
  - GameDriver

DisneyDragon
- extends
- implements
- uses
- creates

DisneyBFly
- extends
- implements
- uses
- creates

DisneyPrincess
- extends
- implements
- uses
- creates

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

[Diagram showing the hierarchy of sprites with arrows indicating extends, uses, implements, and creates relationships]
Instantiating Objects: Factory

- GameDriver now creates \textit{factory}:

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

- Methods use \textit{factory} to create objects:

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

Person Hierarchy with Factory

![Person Hierarchy Diagram]

- Extends: `Person` to `Student` to `Faculty` to `Staff` to `OSUStaff` to `UMStaff` to `Provost`
- Uses: `PeopleFactory` to produce various Person objects
- Implements: `OSUStudent` to `UMStudent`, `OSUProf` to `UMProf`, `OSUSTaff` to `UMStaff`, `OSUDirectory` to `UMDirectory`
Accessory Hierarchy with Factory

Warden and Prisoners
Alternative: Factory Method

- 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

```
extends uses implements creates
```

```
DisneyDragon
MagicDragon
DisneyBFly
MagicBFly
```

```
Sprite
Dragon
Butterfly
Princess
```

```
GameDriver
MagicGame
DisneyGame
```

```
MagicPrincess
DisneyPrincess
```

```
MagicBFly
DisneyBFly
```

```
```

Recall Basic JUnit Recipe

- 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

```
@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

```java
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

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

    @Test
    public void doesSumAccurately() {
        assertEqual(7, p1.add(3, 4));
    }
}
```
JUnit with Factory Methods

- 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

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

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

- 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

- 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