Iteration 3

- Functional
  - Iteration 2 requirements functional (30%)
    - Maintain your bug report
  - Refactoring (20%)
    - Replace temp (i.e. use function call instead of temp var)
    - Rename with “better” name (2 instances)
    - Extract method (i.e. create helper function)
  - No Warnings during compilation (-Werror) (5%)
  - File names and locations, tag, merge (5%)

Iteration 3

- Documentation
  - Peer Reflection (20%)
    - Exchange html + UML with someone (or within group)
    - Complete the worksheet
    - Share with author
  - Doxygen (10%)
  - Bug Report (5%)
  - Cpplint error free EXCEPT rand (5%)

Factory Pattern

(They are classes that create or construct something)

OOP Languages and Factories

- Factories:
  - construct and return an instance of a class type.
  - provide an encapsulation of code required to render an instance of an abstract class type as an implementation class
  - can initialize, pull in data, configure, set state, and perform nearly any other creational operation needed for a class

When do you use a Factory?

- If you have a lot of creational logic for instances of a class type strewn throughout your code base.
- Could you consolidate this logic into one place for uniformity and maintainability? This implies a factory pattern.

When else?

- Might be useful also if you want to limit and define the accepted member implementations of a certain abstract class type for a particular logic flow, but do not want to define at compile time the logic for which type might be used.
  - Example: you might pass in a key to the factory method instead of using a Boolean if…then…else statement to determine the class you want to construct and return from your factory.
    - Frees you from having to use compiled Boolean statements and instead allows for something outside of your factory to determine the logical flow of which implementation of a class type is rendered.

- In other words:
  - If (robot_type) { entities_.push_back(new Robot); }
  - Else if (player_type) {entities_.push_back(new Player); }
  - ...
- INSTEAD
  - Entities_.push_back(Factory->Create(entity_type))

Details…

- Factories can be static for creation only, or repositories, which both create and store references to the created products.
- Factories provide a way to house the creational aspects of a class type.
  - Think about position for entities.

Components of a Factory Pattern

- The Factory: the factory class is the class that renders the product class
- The Product: the product class contains the data or functionality and is part of a series of class types that can be instantiated from a factory method as an instance of an abstract type

Factory UML

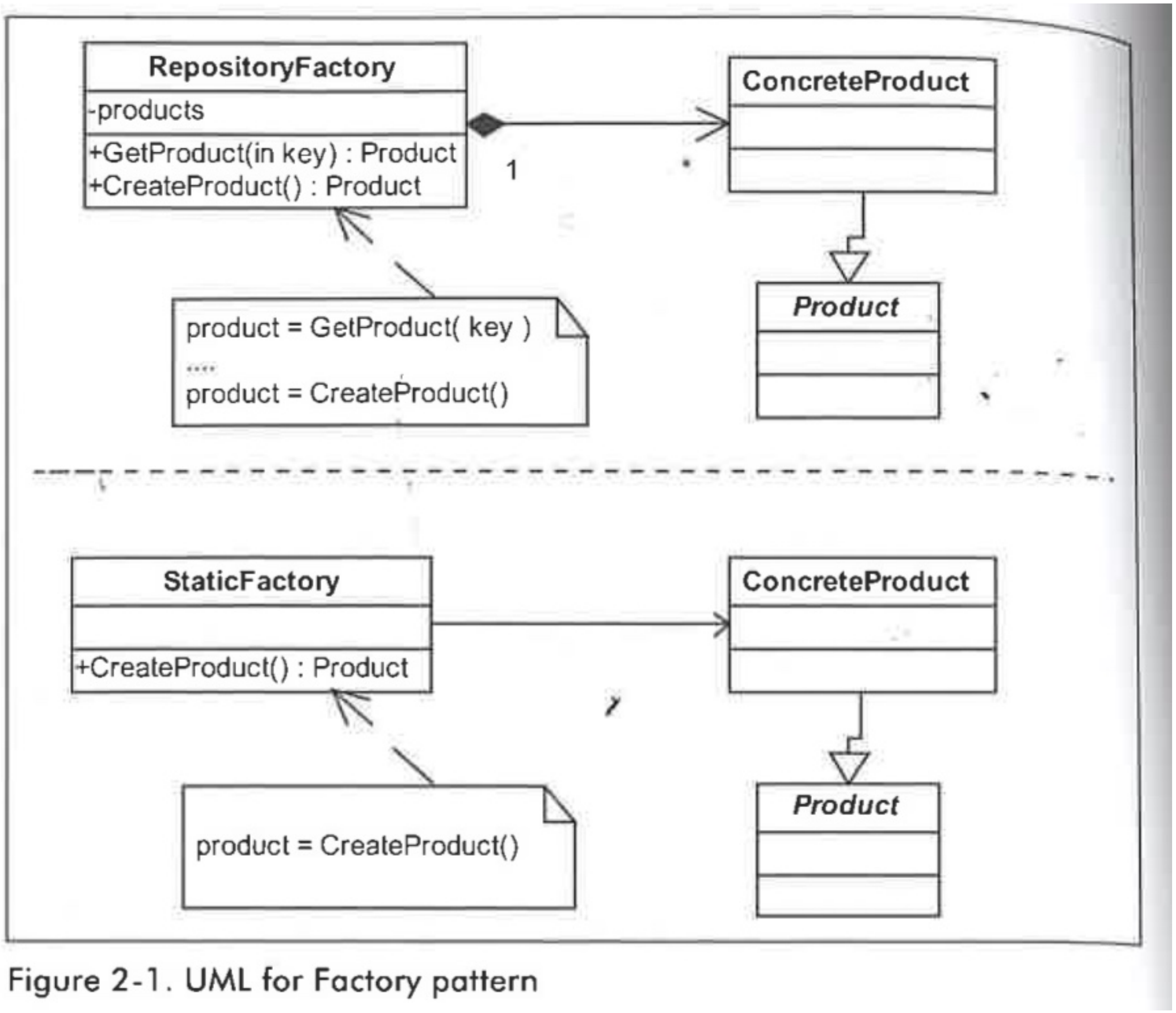
RepositoryFactory
-products
+GetProduct(in key) : Product
+CreateProduct() : Product
1
*
ConcreteProduct
Product
product = GetProduct(key)
....
product = CreateProduct()
StaticFactory
+CreateProduct() : Product
ConcreteProduct
Product
product = CreateProduct()

Figure 2-1. UML for Factory pattern

Problems and Solutions

- Problem 1: Implementation classes with a common base are created in multiple places and no uniformity exists between creational logic.
- Solution 1: Use a factory to encapsulate the class creation code in one place.

Problem 1 Code Example

```
class Wardrobe {
    void AddSuit( suitType) {
        Suit suit;
        if (suitType==SuitType.Armani)
            suit = new Armani();
        else if(suitType == suitType.StripedBusinessSuit)
            suit = new StripedBusinessSuit();
        ... }

class Inventory {
    void AddSuit( suitType) {
        Suit suit; ... }
```

Looking at the Solution

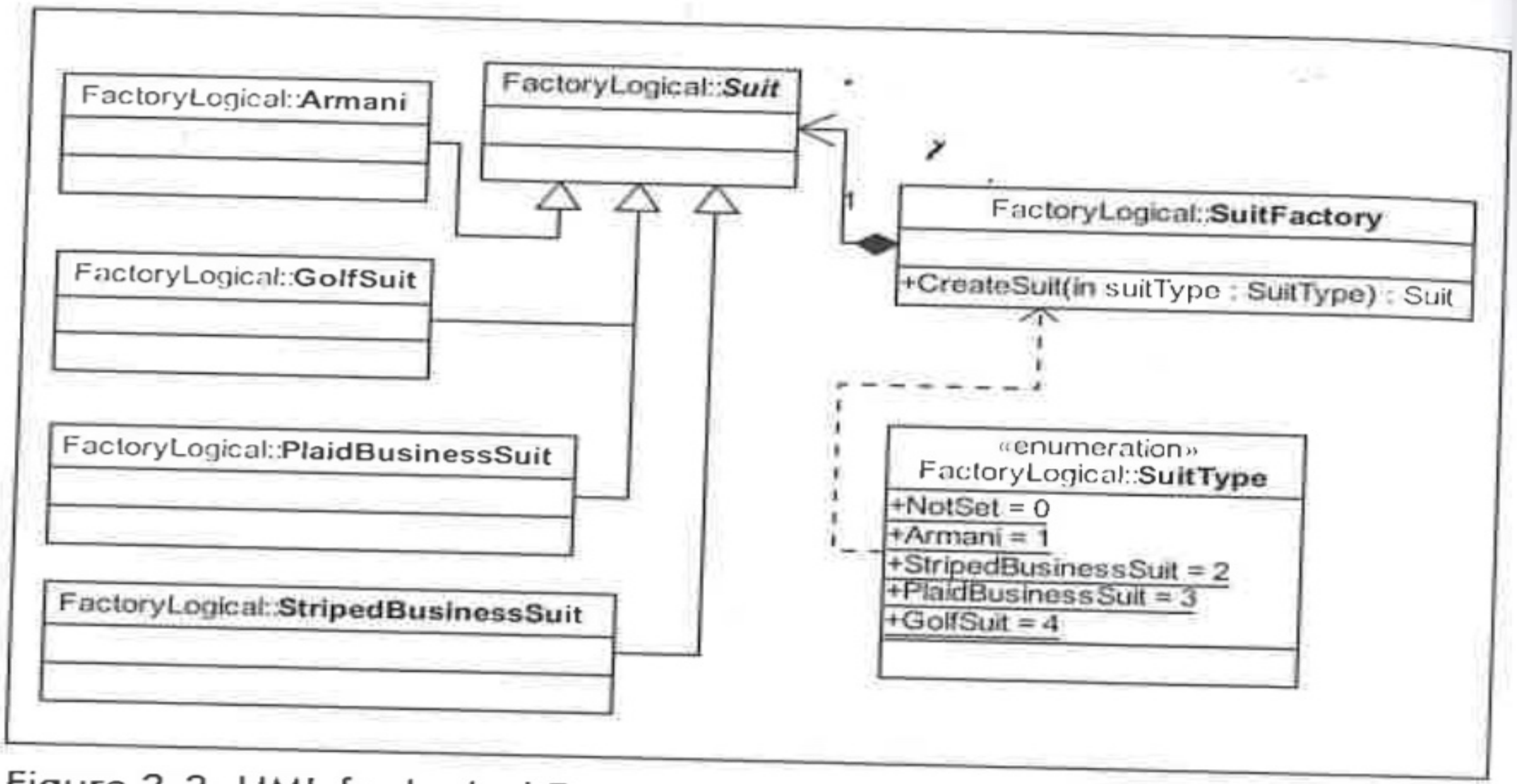

Figure 2-2. UML for logical Factory pattern example

Solution: Use a factory to encapsulate the class creation code in one place

```
class SuitFactory
{
    Suit suit;
    if (suitType==SuitType.Armani)
        suit = new Armani();
    else if (suitType==SuitType.StripedBusinessSuit)
        suit = new StripedBusinessSuit();
    else if (suitType==SuitType.PlaidBusinessSuit)
        suit = new PlaidBusinessSuit();
    else if (suitType == SuitType.GolfSuit)
        suit = new GolfSuit();
    return suit;
}
```

Clean up also…

```
class Wardrobe {
    SuitFactory factory = new SuitFactory();
     void AddSuit( suitType) {
        Suit suit = factory.CreateSuit(suitType);
```

Problem 2

- Problem 2: Class types to be created are determined by immutable logical code and extending the returnable types requires scale modifications to the factory.
- Solution 2a: Replace conditional logic with a class activator, using reflection to determine the needed types for different workflows.

We need to…

- expand the logical if…then…else code.
- This presents a scalability issue, due to the fact that we are limited within the design to only the types included within the conditional logic, and cannot readily expand the accepted types without modifying logic.

Initial Code

```
class SuitFactory
{
        Suit suit;
        if (suitType==SuitType.Armani)
                suit = new Armani();
        else if (suitType==SuitType.StripedBusinessSuit)
                suit = new StripedBusinessSuit();
        else if (suitType==SuitType.PlaidBusinessSuit)
                suit = new PlaidBusinessSuit();
        else if (suitType == SuitType.GolfSuit)
                suit = new GolfSuit();
        else if (suitType ==SuitType.MotorcycleSuit)
                suit = new MotorcycleSuit();
        else if (suitType==SuitType.JoggingSuit)
                suit = new JoggingSuit();
        else if (suitType == SuitType.LadiesPantSuit)
                suit = new LadiesPantSuit();
        else if (suitType ==SuitType.SolidBusinessSuit)
                suit = new SolidBusiness Suit();
        else if (suitType==SuitType.TennisSuit)
                suit = new TennisSuit();

        return suit;
}
```

Solution: Pass the factory

```
class SuitFactory {
    virtual suit CreateSuit() {}
}

class ArmaniFactory (SuitFactory) {
    suit CreateSuit() {
        return new Armani();
}

class Wardrobe {
    void AddSuit( SuitFactory) {
        Suit suit = SuitFactory.CreateSuit();
        ...
```

What Did We Do

- Created a base factory.
- Created a factory for each subtype.
- Passed the factory instead of the enumerated type to get the required object.

YOUR TASK – At your white board ...

- Draw UML for factory for Robot and Player.
- Do you pass in params or do they reside in the factory??
- Use the factory to store potential positions of entities and set the position that way.
- Sketch out how this changes initialization of these entities.