

Creational Design Patterns

Part 2

Creational Design Patterns

- Singleton
- Factory Method
- Abstract Factory
- Builder
- Prototype



Creational Patterns: Factory Method

- Suppose we are building a registrar system for Western...
 - Example from Joanne Atlee, University of Waterloo

Creational Patterns: Factory Method

Registrar.cpp

```
void Registrar::admitStudent(const string& name, const string& dept)
{
    Student *s;

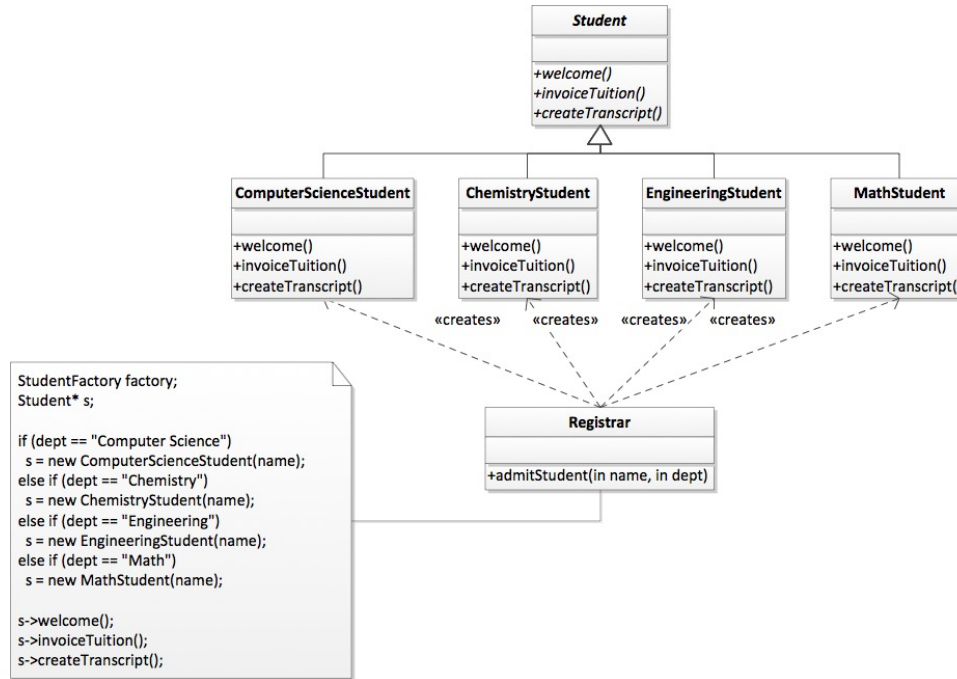
    // Instantiate a concrete object -- violate 'program to an
    // interface, not an implementation'
    if (dept == "Computer Science")
        s = new ComputerScienceStudent(name);
    else if (dept == "Chemistry")
        s = new ChemistryStudent(name);
    else if (dept == "Engineering")
        s = new EngineeringStudent(name);
    else if (dept == "Math")
        s = new MathStudent(name);

    // ...
    cout << "Admitting student " << s->name() << endl;
    // Each student type has its own admission operations

    s->welcome();
    s->invoiceTuition();
    s->createTranscript();

    cout << endl;
}
```

Creational Patterns: Factory Method



Creational Patterns: Factory Method

- Problems:
 - Each time we use `new`, we violate the “Program to an interface, not an implementation” design principle
 - Tying code to a concrete implementation in this fashion makes it fragile and less flexible; harder to reuse
 - By coding to an interface instead, our code would work with new classes implementing that interface
 - Furthermore, we have to violate the Open-Closed Principle each time we add a new department

Creational Patterns: Factory Method

- Toward a solution: encapsulate what varies

StudentFactory.cpp

```
Student* StudentFactory::createStudent(const string& name, const string& dept)
{
    Student *s;

    // Instantiate a concrete object -- violate 'program to an
    // interface, not an implementation'
    if (dept == "Computer Science")
        s = new ComputerScienceStudent(name);
    else if (dept == "Chemistry")
        s = new ChemistryStudent(name);
    else if (dept == "Engineering")
        s = new EngineeringStudent(name);
    else if (dept == "Math")
        s = new MathStudent(name);

    // ...

    return s;
}
```

Creational Patterns: Factory Method

Registrar.cpp

```
void Registrar::admitStudent(const string& name, const string& dept)
{
    Student *s;
    StudentFactory factory;

    s = factory.createStudent(name, dept);

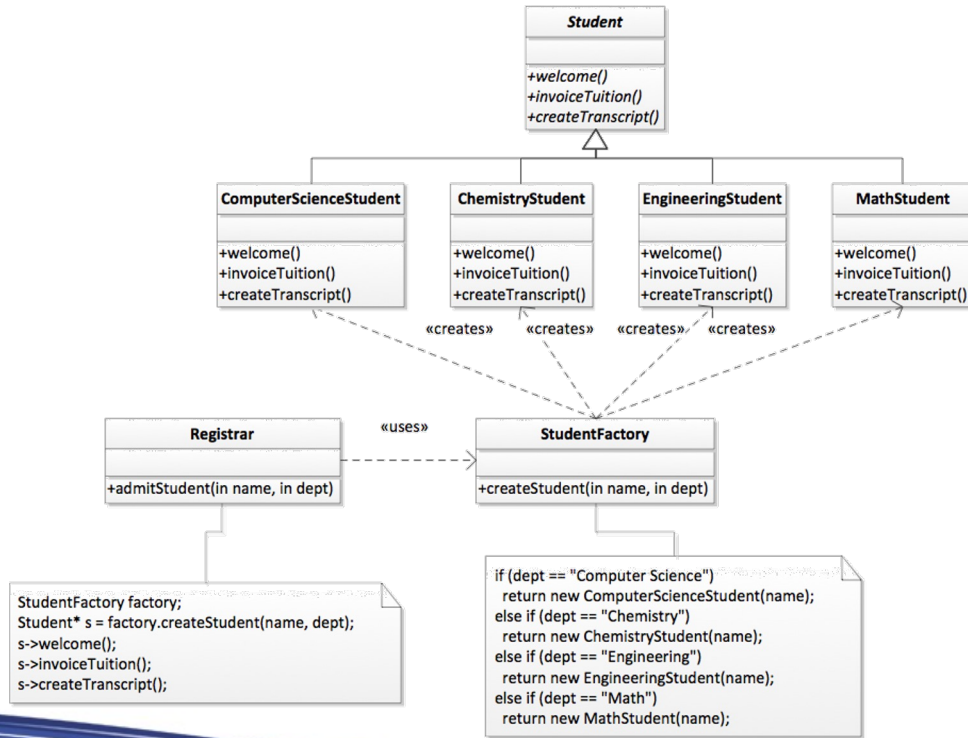
    cout << "Admitting student " << s->name() << endl;

    // Each student type has its own admission operations

    s->welcome();
    s->invoiceTuition();
    s->createTranscript();

    cout << endl;
}
```


Creational Patterns: Factory Method



Creational Patterns: Factory Method

- This is called a *Simple Factory* – not a design pattern
 - Keep in mind that `StudentFactory` may have many clients
 - We might also have other classes that need to create students
 - This encapsulates `Student` creation in one class so we only have to make changes in one place when new `Student` types added
 - This also decouples `Registrar` from concrete implementations, making it much more reusable

Creational Patterns: Factory Method

- Problems with this Simple Factory:
 - We've just offloaded the problem to a new class; instead of high coupling between `Registrar` and the various classes, we now have high coupling between `StudentFactory` and the `Student` classes
 - Still have to violate the Open-Closed Principle when we want to add new `Student` types to `StudentFactory`
 - The `if-else` block is unwieldy
 - Using `strings` as parameters is error-prone

Creational Patterns: Factory Method

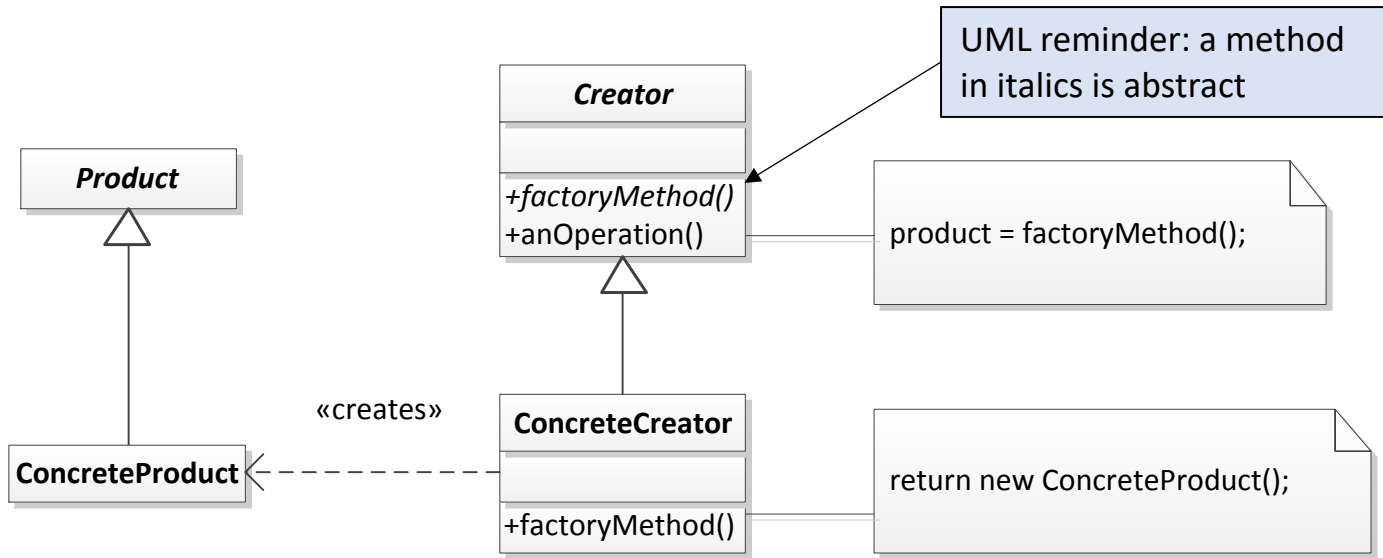
Design Pattern: Factory Method

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

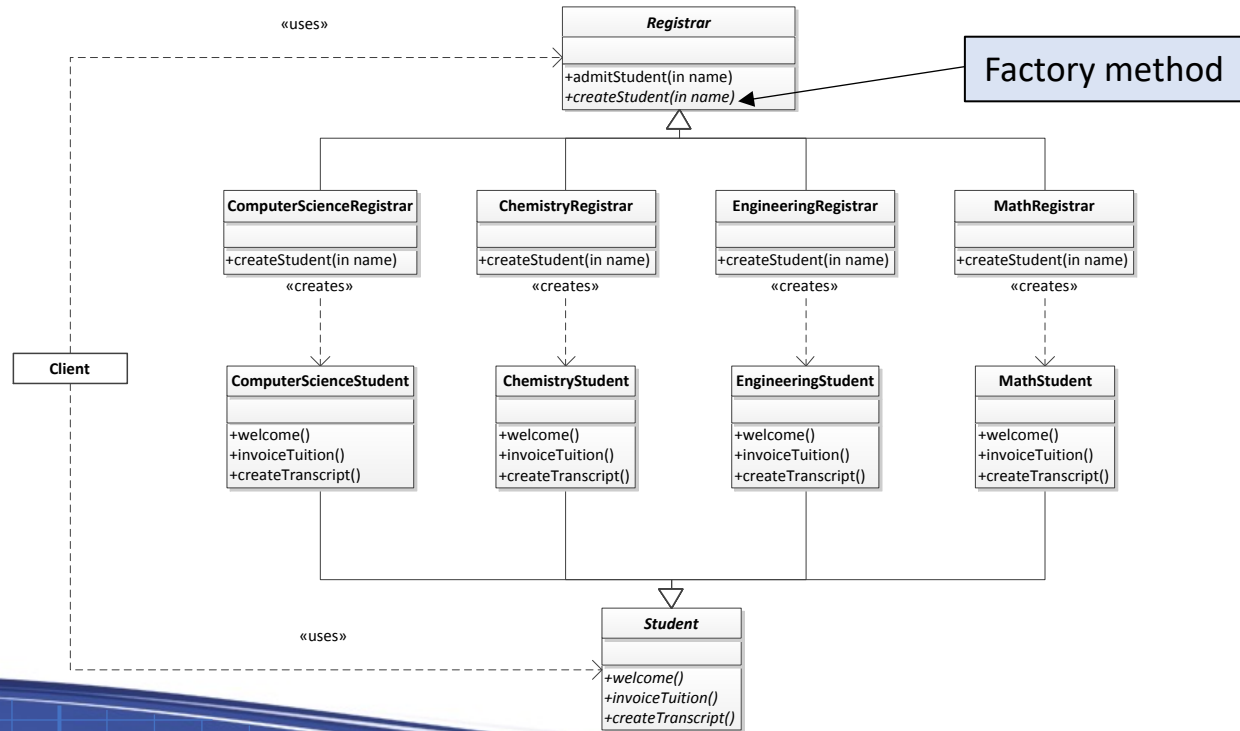
Creational Patterns: Factory Method

- Applicability:
 - A class can't anticipate the class of objects it must create
 - A class wants its subclasses to specify the objects it creates

Creational Patterns: Factory Method



Creational Patterns: Factory Method



Creational Patterns: Factory Method

Registrar.h

```
class Registrar
{
    public:
        void admitStudent(const std::string& name);

    protected:
        virtual Student* createStudent(const std::string& name) = 0;
};
```


Creational Patterns: Factory Method

Registrar.cpp

```
void Registrar::admitStudent(const string& name)
{
    Student *s = this->createStudent(name);

    cout << "Admitting student " << s->name() << endl;

    // Each student type has its own admission operations
    s->welcome();
    s->invoiceTuition();
    s->createTranscript();

    cout << endl;
}
```

Creational Patterns: Factory Method

ComputerScienceRegistrar.cpp

```
class ComputerScienceRegistrar : public Registrar
{
    public:
        virtual Student* createStudent(const std::string& name)
        {
            return new ComputerScienceStudent(name);
        }
};
```

Creational Patterns: Factory Method

main.cpp

```
void enrollStudents(map<string, Registrar*>& registrars, map<string, string> studentsToEnroll)
{
    for (map<string, string>::iterator it = studentsToEnroll.begin(); it != studentsToEnroll.end(); ++it)
    {
        Registrar* registrar = registrars[it->second];
        registrar->admitStudent(it->first);
    }
}

int main()
{
    // Still have to hard-code concrete classes somewhere
    // But, we'll use Registrar and Student throughout our
    // code as much as possible -- see enrollStudents()
    map<string, Registrar*> registrars;
    registrars["cs"] = new ComputerScienceRegistrar();
    registrars["eng"] = new EngineeringRegistrar();
    registrars["math"] = new MathRegistrar();

    map<string, string> studentsToEnroll;
    studentsToEnroll["Jeff"] = "cs";
    studentsToEnroll["Bob"] = "eng";
    studentsToEnroll["Jane"] = "math";

    enrollStudents(registrars, studentsToEnroll);
}
```

Creational Patterns: Factory Method

Another example:

- Suppose we are creating a game with various levels
- We have a `GameLevel` class and a `Monster` class
- Each level will have specific monsters
 - Fire monsters on fire levels, ice monsters on ice levels, electric monsters on electric levels, etc.
- `GameLevel` is a client, and it uses `Monster` products



Creational Patterns: Factory Method

```
class GameLevel
{
    public:
    GameLevel()
    {
        // Create the level
        ...
        // Create monsters for the level
        ...
        // Add the monsters to the level
        ...
    }
};
```

Creational Patterns: Factory Method

- Solution 1: Use `if-else` everywhere we need to create a `Monster`

```
Monster* m;  
  
if (isFireLevel)  
{  
    m = new FireMonster();  
}  
else if (isIceLevel)  
{  
    m = new IceMonster();  
}  
else  
{  
    m = new RegularMonster();  
}
```

Creational Patterns: Factory Method

- Solution 2: Move `if-else` inside a special method

```
Monster* createMonster()  
{  
    if (isFireLevel)  
    {  
        return new FireMonster();  
    }  
    else if (isIceLevel)  
    {  
        return new IceMonster();  
    }  
    else  
    {  
        return new RegularMonster();  
    }  
}
```

Creational Patterns: Factory Method

- The factory method is solution 2, with a twist
 - `createMonster` function is protected
 - `FireGameLevel` and `IceGameLevel` will overload it
 - Will change the monsters used in the `GameLevel`

Creational Patterns: Factory Method

```
class GameLevel
{
public:
    GameLevel()
    {
        // Create the level
        ...
        // Create monsters for the level
        Monster* m1 = createMonster();
        Monster* m2 = createMonster();
        // Add the monsters to the level
        ...
    }
    ...
protected:
    // Can provide a default implementation
    virtual Monster* createMonster()
    {
        return new RegularMonster();
    }
};
```

Creational Patterns: Factory Method

```
class FireGameLevel : public GameLevel
{
    public:
        // inherits the constructor
    protected:
        virtual Monster* createMonster()
        {
            return new FireMonster();
        }
};
```

Creational Patterns: Factory Method

```
class IceGameLevel : public GameLevel
{
    public:
        // inherits the constructor
    protected:
        virtual Monster* createMonster()
        {
            return new IceMonster();
        }
};
```

Creational Patterns: Factory Method

- Consequences:
 - Factory methods eliminate the need to bind application-specific classes into our code
 - The code only deals with the `Product` interface, so it can work with any user-defined `ConcreteProduct` classes
 - Our `Registrar` only deals with the `Student` interface, so it can work with any user-defined concrete student classes
 - Clients have to subclass the `Creator` class just to create a particular `ConcreteProduct` object