Creational Design Patterns

Creational Design Patterns

Two main goals:

1. Encapsulate knowledge about which concrete classes the system uses

2. Hide how instances of these classes are created and built

Creational Design Patterns

- System at large knows about objects through their interfaces defined by abstract classes

- Give us flexibility in:
  - *what* gets created
  - *who* creates it
  - *how* it gets created
  - *when* it gets created

Creational Design Patterns

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

Creational Patterns: Singleton

- Consider a class called `Logger`
    - Logs information to a file
    - Needed by many different parts of an application

Creational Patterns: Singleton

Logger.h

```
class Logger
{
   public:
      Logger();
      virtual ~Logger();
      const Logger& log(const std::string& message) const;
      const Logger& operator<<(const  std::string&  message) const;

   private:
      mutable  std::ofstream  _output;
};
```

Creational Patterns: Singleton

Logger.cpp

```
Logger::Logger()
{
   this->_output.open("program.log");
}

Logger::~Logger()
{
   this->_output.close();
}

const Logger& Logger::log(const string& message) const
{
   this->_output   <<   message   << endl;
   return *this;
}

const Logger& Logger::operator<<(const string& message) const
{
   return this->log(message);
}
```

Creational Patterns: Singleton

main.cpp

```
void f(const Logger& log)
{
   log << "In function f()";
}

int main()
{
   Logger log;
   log << "Starting program";

   f(log);
}
```

Creational Patterns: Singleton

Output

```
$ ./main
$ cat program.log
Starting program
In  function f()
```

Creational Patterns: Singleton

- As our application grows, we will want to have logging in more and more functions
- Potential solutions:
  - Pass around a `Logger` object to the functions that need it
  - Create a new `Logger` object in each function that needs it
  - Use a global `Logger` object that all functions can access from anywhere

Creational Patterns: Singleton

- Suppose we opt to pass around a `Logger` object
- Later, we add a `Person` class
- Each `Person` has a `Car`

Creational Patterns: Singleton

Person.h

```
class Person
{
   public:
      Person(const std::string& name);
      virtual ~Person();
      Car* car() const;

   private:
      std::string _name;
      Car* _car;
};
```

Creational Patterns: Singleton

Person.cpp

```
Person::Person(const std::string& name)
{
   this->_name = name;
   this->_car = new Car();
}

Person::~Person()
{
   delete this->_car;
}

Car* Person::car() const
{
   return this->_car;
}
```

Creational Patterns: Singleton

Car.h

```
class Car
{
   public:
      Car();

      void turnOn();
      void turnOff();
};
```

Creational Patterns: Singleton

- Now we want to add logging so that a log entry is created each time a person's `Car` is turned on or off

- Which class(es) do we need to modify?

Creational Patterns: Singleton

Person.h

```
class Person
{
   public:
      Person(const std::string& name, const Logger& log);
      virtual ~Person();
      Car* car() const;

   private:
      std::string _name;
      Car* _car;
};
```

Creational Patterns: Singleton

Person.cpp

```
Person::Person(const std::string& name, const Logger& log)
{
   this->_name = name;
   this->_car = new Car(log);
}

Person::~Person()
{
   delete this->_car;
}

Car*  Person::car()  const
{
   return this->_car;
}
```

Creational Patterns: Singleton

Car.h

```
class Car
{
   public:
      Car(const Logger& log);

      void turnOn();
      void turnOff();

   private:
      const Logger* _log;
};
```

Creational Patterns: Singleton

Car.cpp

```
Car::Car(const Logger& log) : _log(log)
{
}

void Car::turnOn()
{
   this->_log  <<  "Turning  on car";
}

void Car::turnOff()
{
   this->_log  <<  "Turning  off car";
}
```

Creational Patterns: Singleton

main.cpp

```
int main(){

   Logger log;
   Person p("Joe", log);

   log << "Starting program";

   // Side note: what design principle has been violated here?

   Car* car = p.car();
   car->turnOn();
   car->turnOff();
}
```

Creational Patterns: Singleton

- What are the problems with this solution?

- What if, instead, we created a new `Logger` object in every function that needed logging?

Creational Patterns: Singleton

Logger.cpp

```
Logger::Logger()
{
   this->_output.open("program.log");
}

Logger::~Logger()
{
   this->_output.close();
}

const Logger& Logger::log(const string& message) const
{
   this->_output   <<   message   << endl;
   return *this;
}

const Logger& Logger::operator<<(const string& message) const
{
   return this->log(message);
}
```

- Any issues with this?

Creational Patterns: Singleton

- What if, instead, we used a global variable that all functions could access?

```
const  Logger*  const  globalLogger  =  new Logger();
```

```
void f()
{
   *globalLogger << "In function f()";
}
```

```
void Car::turnOn()
{
   *globalLogger << "Turning on car";
}
```

- Problems?

Creational Patterns: Singleton

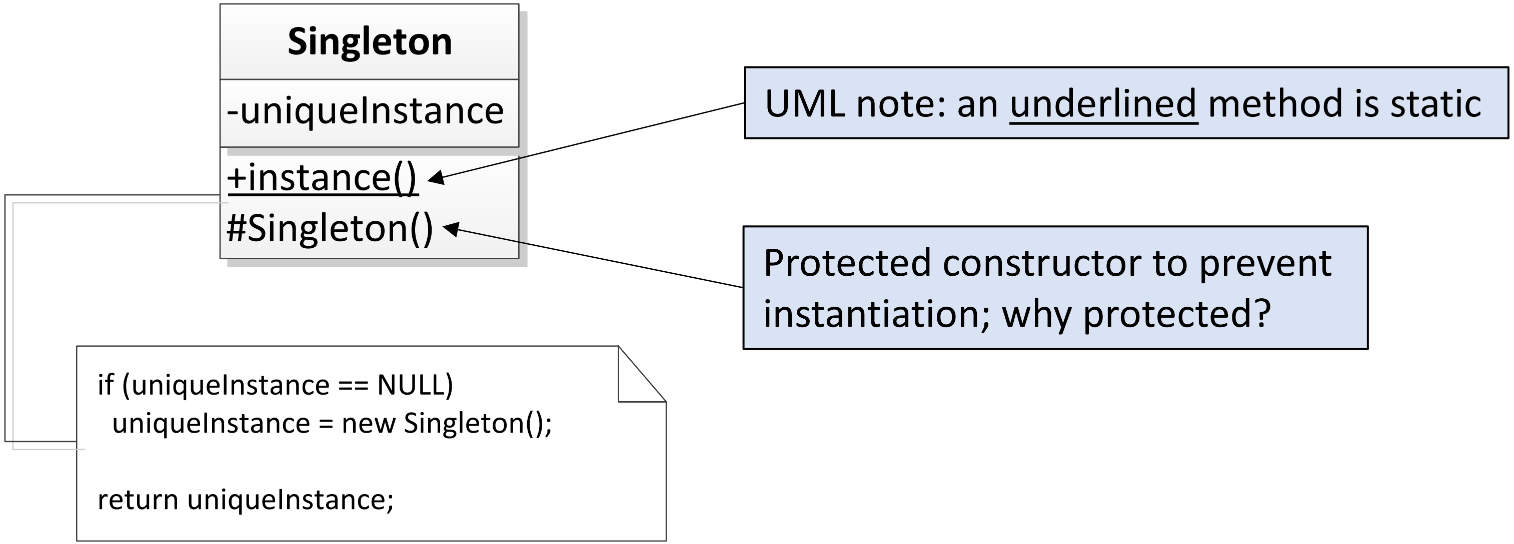
Design Pattern:
Singleton

Ensure a class has only one instance, and provide a global point of access to it.

Creational Patterns: Singleton

- Applicability:
  - There must be exactly one instance of a class
  - It must be accessible to clients from a well-known access point
  - The sole instance should be extensible by subclassing

Creational Patterns: Singleton



Creational Patterns: Singleton

Logger.h

```
class Logger
{
   public:
      virtual ˜Logger();
      static const Logger& instance();
      const Logger& log(const std::string& message)  const;
      const Logger& operator<<(const  std::string&  message)  const;

   protected:
      Logger(); // Prevent instantiation

   private:
      // Prevent copying and assignment
      Logger(const Logger& other) { };
      Logger& operator=(const Logger& other) { };
      mutable std::ofstream _output;
      static const Logger* _instance;
};
```

Creational Patterns: Singleton

Logger.cpp

```
const Logger* Logger::_instance = NULL;
const Logger& Logger::instance()
{
   if (_instance  == NULL)
      _instance  =  new  Logger();
   return *_instance;
}
Logger::Logger()
{
   this->_output.open("program.log");
}
Logger::~Logger()
{
   this->_output.close();
}
const Logger& Logger::log(const string& message) const
{
   this->_output   <<   message   << endl;
   return *this;
}
const Logger& Logger::operator<<(const string& message) const
{
   return this->log(message);
}
```

Creational Patterns: Singleton

main.cpp

```
int main(){
   Logger::instance() << "Starting program";

   Person p("Joe");

   Car* car = p.car();

   car->turnOn();
   car->turnOff();
}
```

Creational Patterns: Singleton

- Consequences:
  - Controlled access to sole instance
  - Lazy initialization
  - Reduced name space
  - Permits refinement through subclassing
  - Permits a variable number of instances, if needed
  - Have to worry about who deletes the instance
    - `std::shared_ptr` or `boost::shared_ptr` can help with this

Creational Design Patterns

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

Creational Patterns: Factory Method

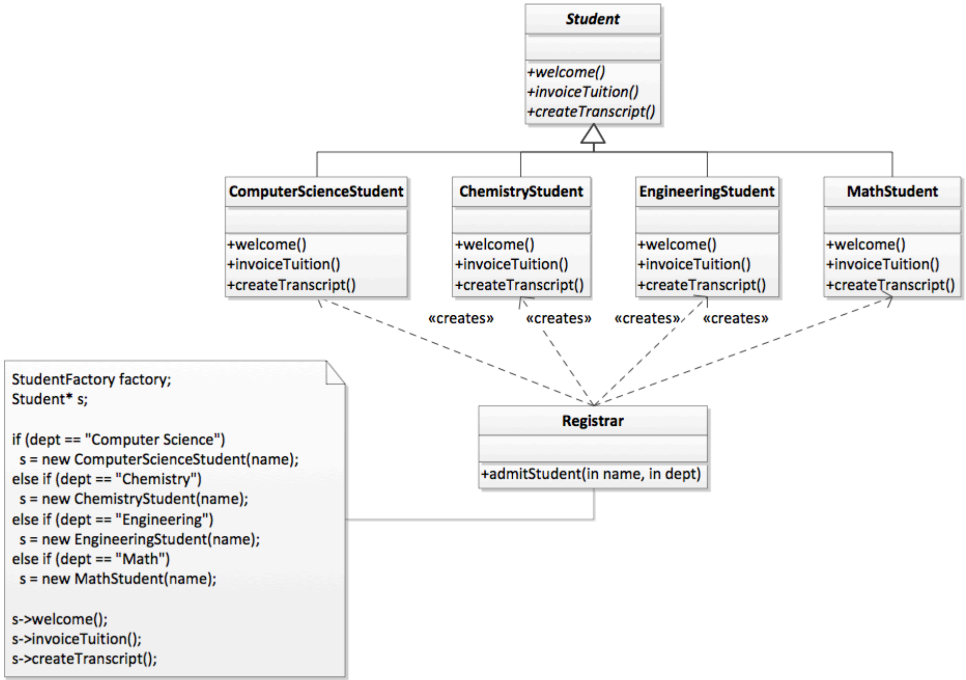
- Suppose we are building a registrar system for Western...
  - Example from Joanne Atlee, University of Waterloo

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

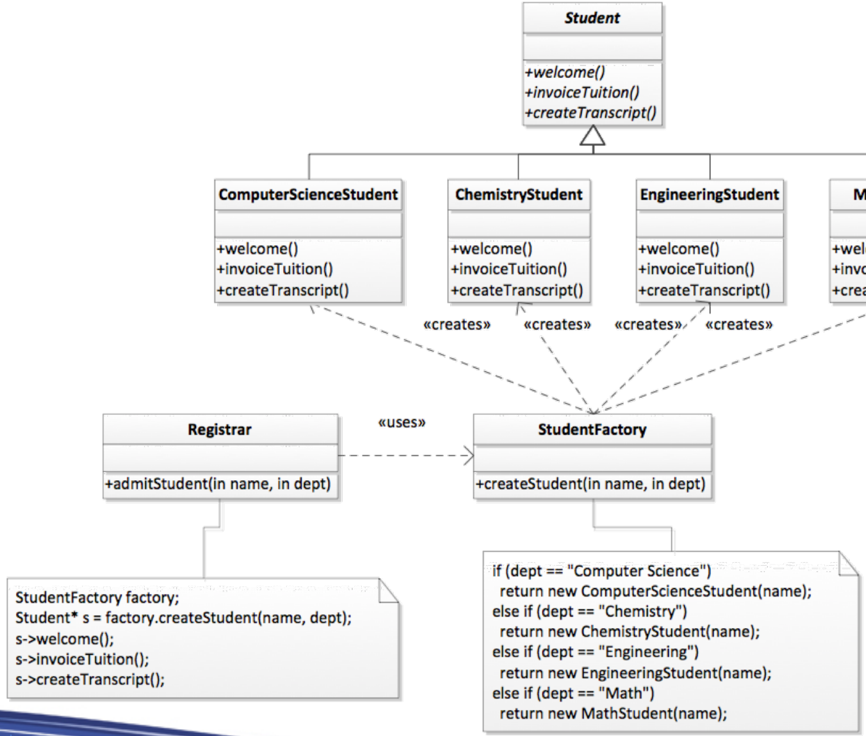
```
Student* StudentFactory::createStudent(const string& name, const string& dept)
{
   Student *s;

   // 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

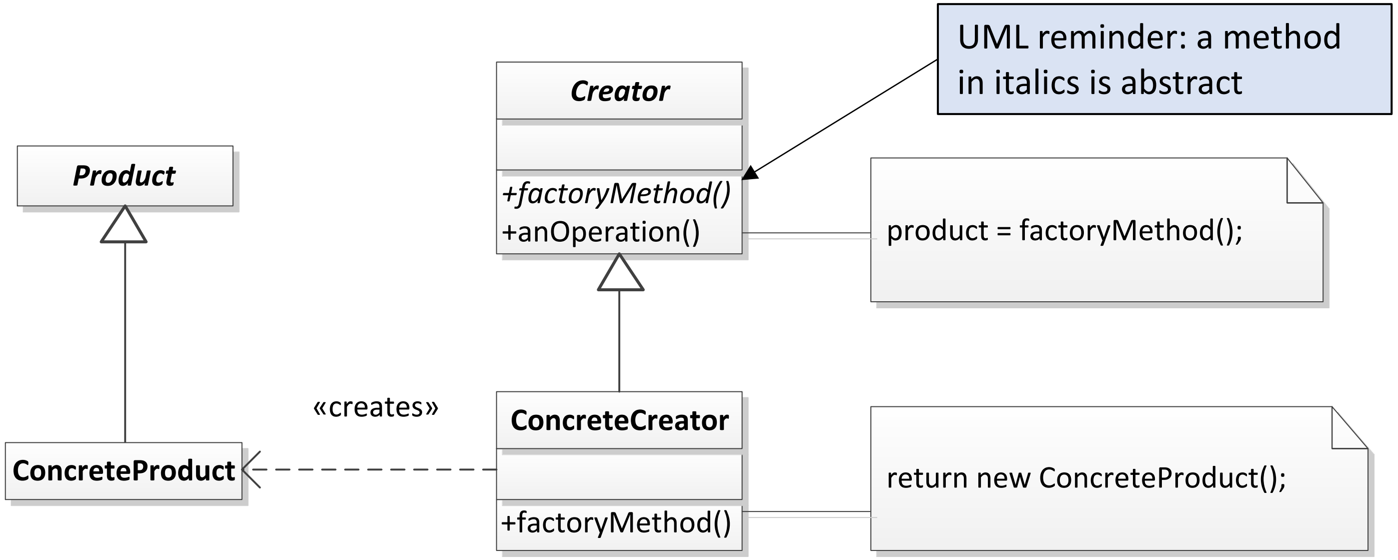
Design Pattern:
Factory Method

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

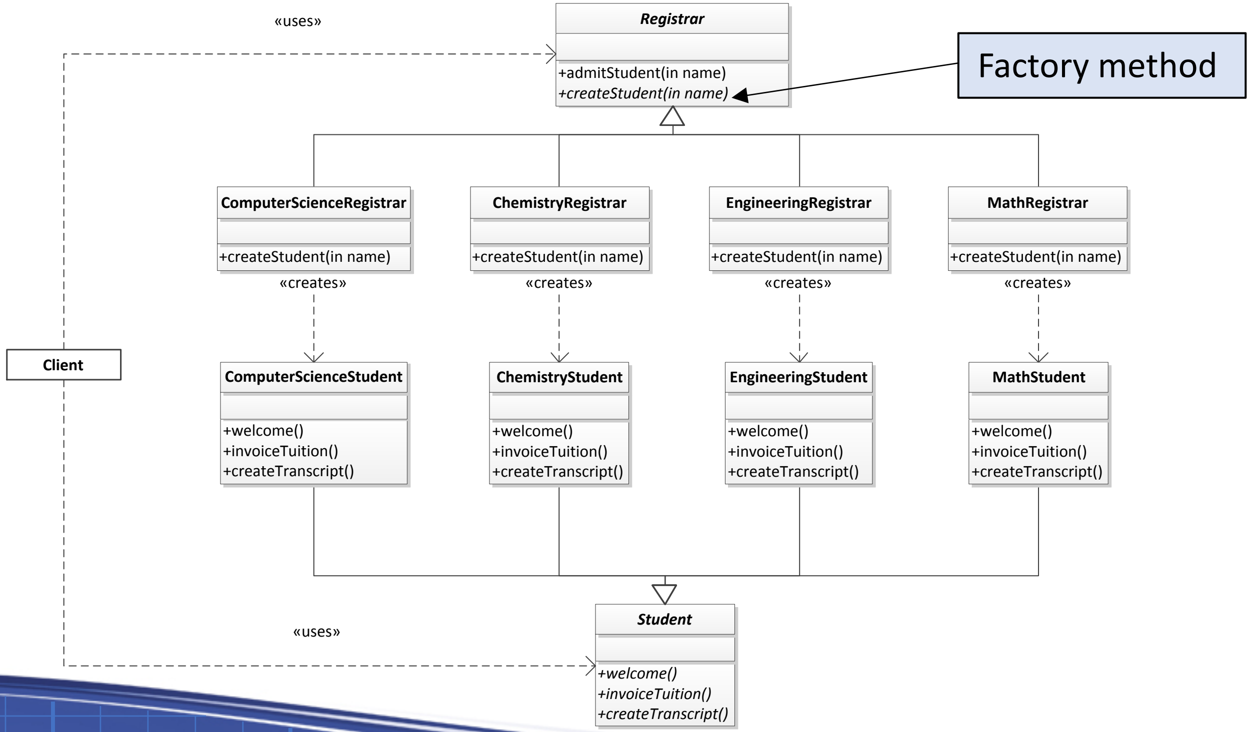
Creational Patterns: Factory Method

- 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

Creational Design Patterns

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

Creational Patterns: Abstract Factory

- Factory method allows us to create one product through inheritance
- Sometimes, we want to create families of related products
- Consider our `GameLevel` classes
  - In addition to specific monsters, we may want levels to have a specific floor, sky, walls, and so on

Creational Patterns: Abstract Factory

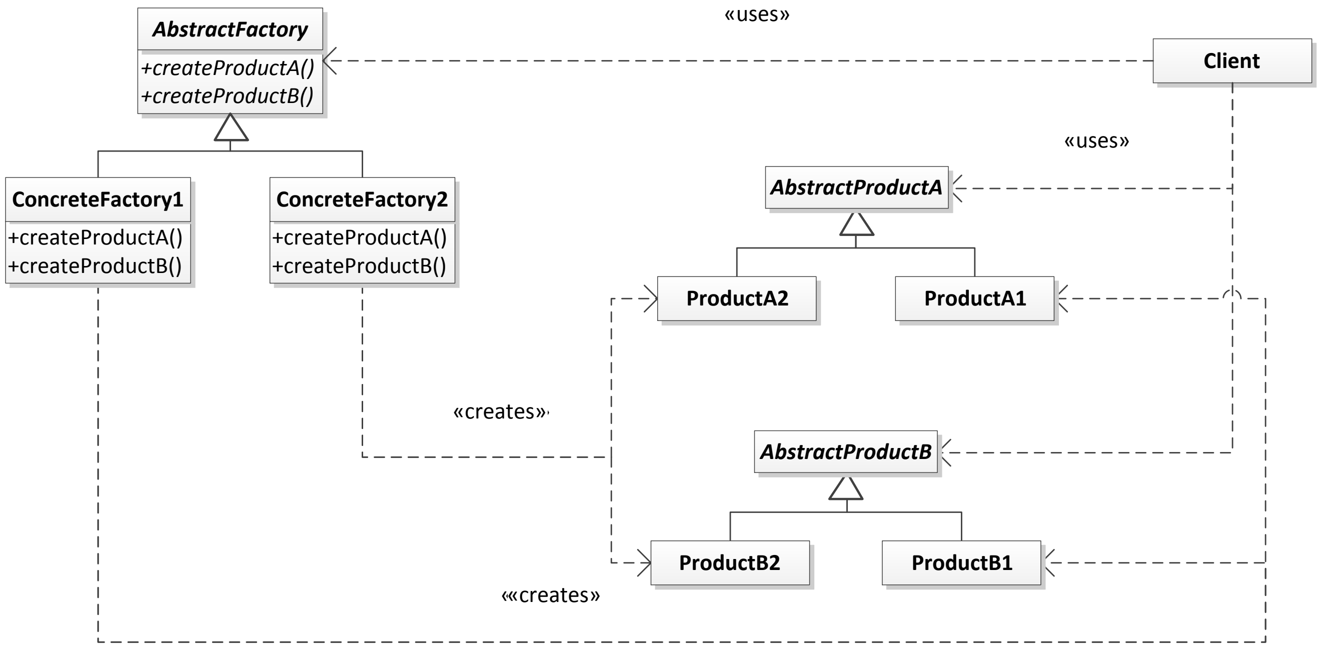
Design Pattern:
Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

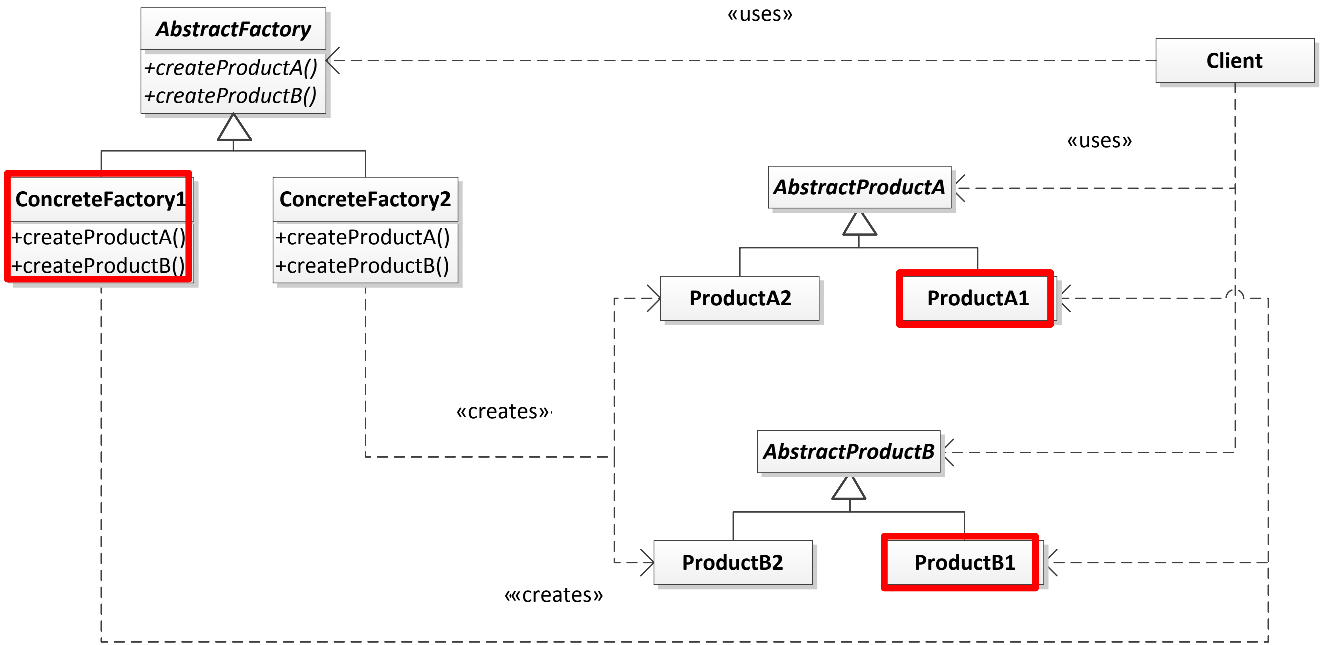
Creational Patterns: Abstract Factory

- Applicability:
  - A system should be independent of how its products are created
  - A system should be configured with one of multiple families of products
  - A family of related product objects are designed to be used together, and you need to enforce this constraint
  - You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations

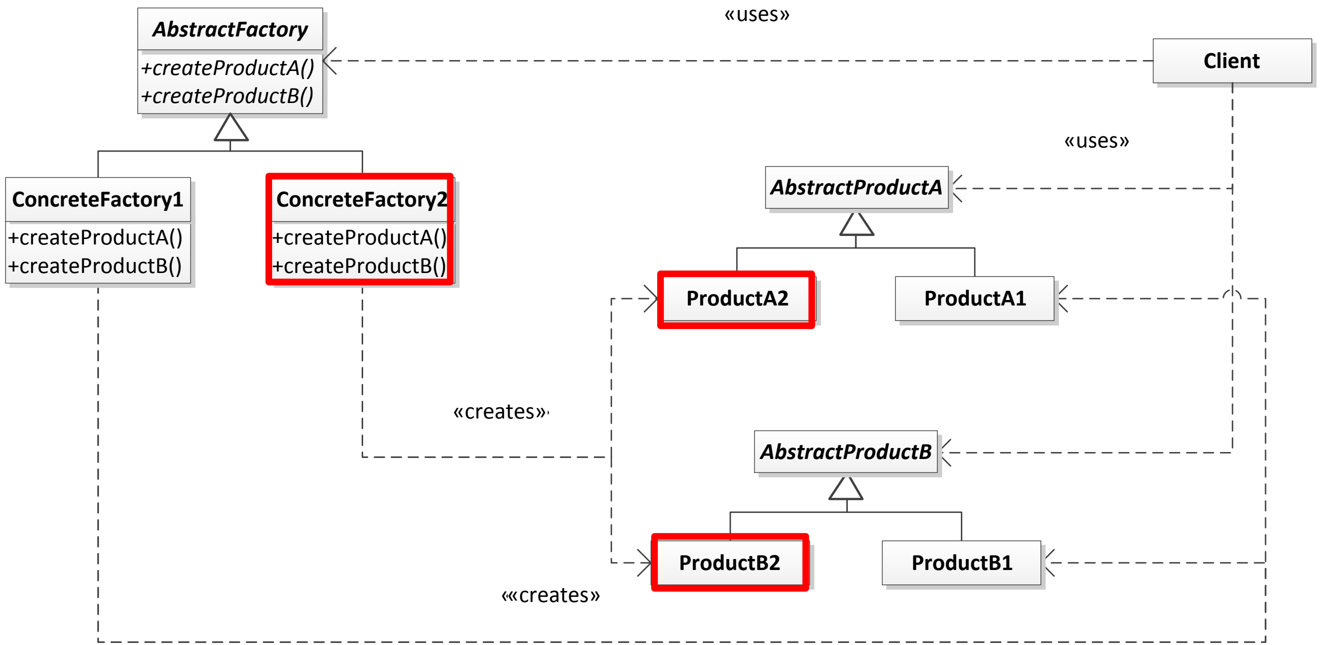
Creational Patterns: Abstract Factory



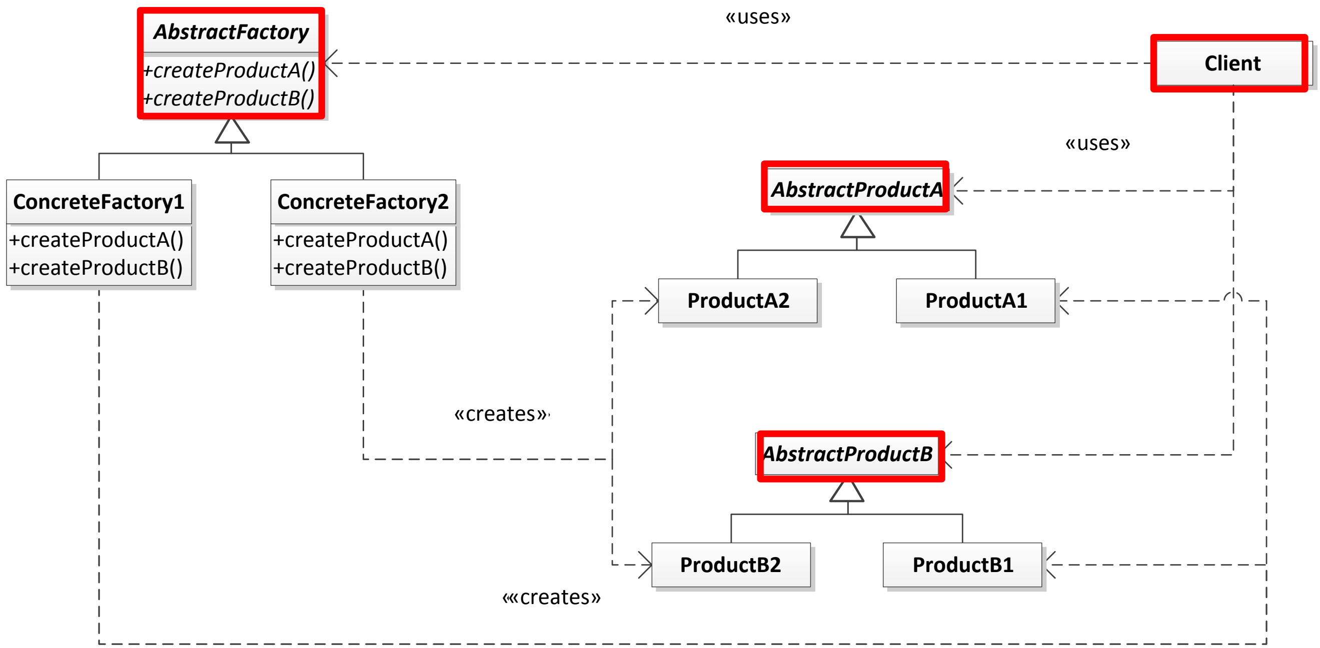
Creational Patterns: Abstract Factory



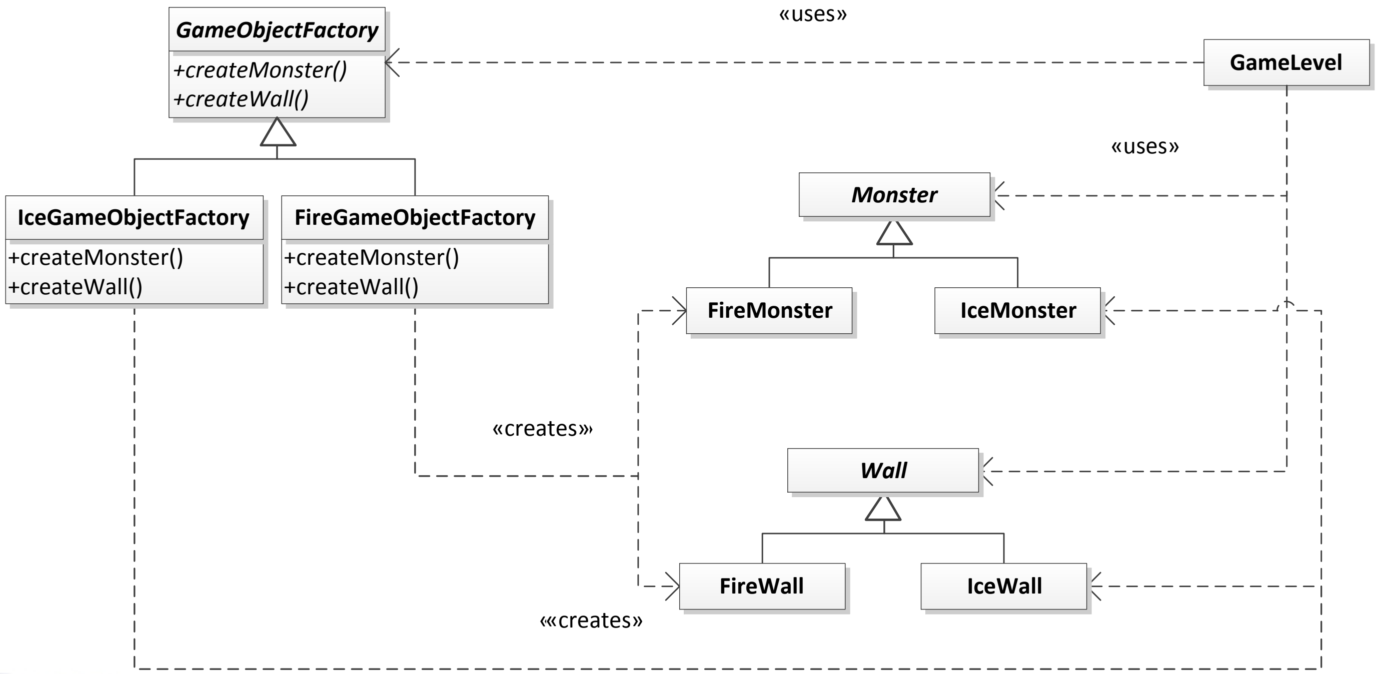
Creational Patterns: Abstract Factory



Creational Patterns: Abstract Factory



Creational Patterns: Abstract Factory



Creational Patterns: Abstract Factory

```
class GameLevel
{
   public:
      GameLevel(GameObjectFactory* factory)
      {
         this->_factory = factory;
         Monster* m1 = factory->createMonster();
         Monster* m2 = factory->createMonster();
         Wall* w1 = factory->createWall();
         // ...
      }
   private:
      GameObjectFactory* _factory;
};
```

Creational Patterns: Abstract Factory

Consequences:

- Isolates concrete classes
  - Client controls when objects are created
  - Factory controls which objects are created and how
- Makes exchanging product families easy
- Promotes consistency among products
- Supporting new kinds of products is difficult

Creational Patterns: Abstract Factory

- Factory Method:
  - Creates a single product
  - Uses inheritance
  - Superclass methods remain generic and use the factory method as needed to create the product
- Abstract Factory:
  - Collects multiple factory methods into a class to create multiple related products
  - Uses aggregation / composition
  - Client remains generic and uses the factory as needed to create the products

Creational Design Patterns

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

Creational Patterns: Builder

- Suppose we are building a new web site for Pizza Pizza

- We have to support two types of pizza:
  - Pre-defined pizzas: Pepperoni and Cheese, Hawaiian, Deluxe, etc.
  - Custom pizzas

Creational Patterns: Builder

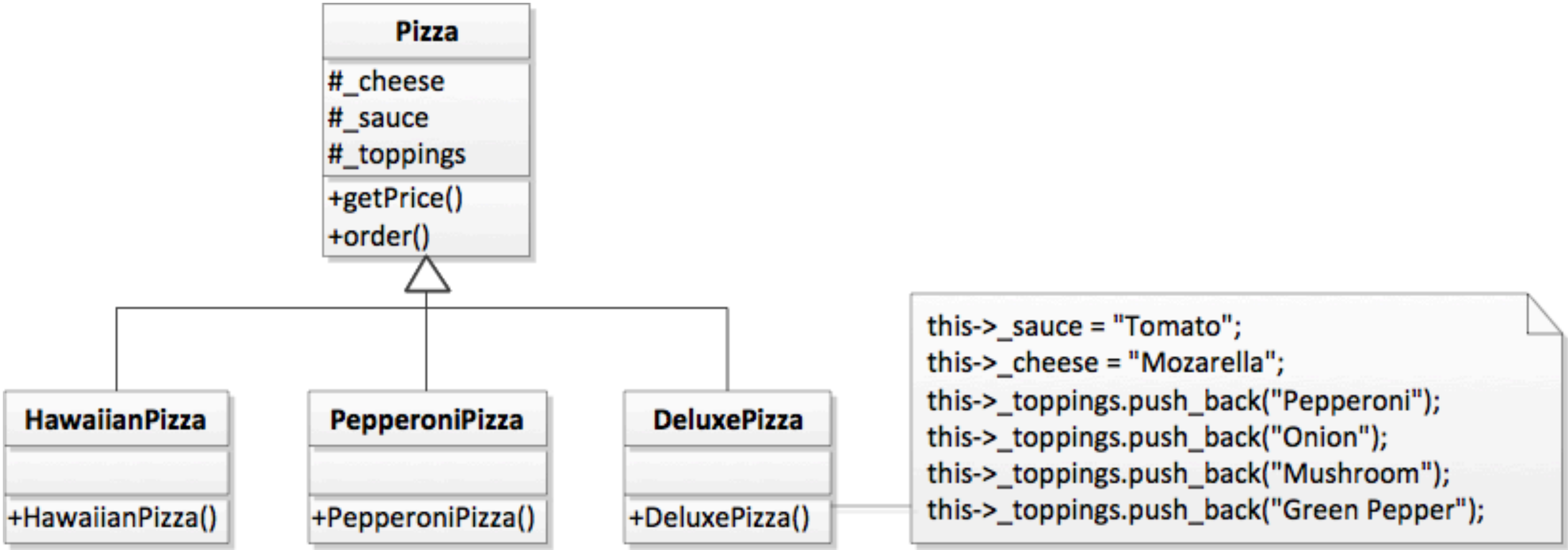
We might have the following code in various places throughout our application:

```
// Build a Hawaiian pizza
Pizza *pizza = new Pizza(12);  // 12" pizza
pizza->addTopping("Pineapple");
pizza->addTopping("Ham");

// ...

// Build a Deluxe pizza
Pizza *pizza = new Pizza(8);
pizza->addTopping("Pepperoni");
pizza->addTopping("Mushroom");
pizza->addTopping("Green Peppers");
pizza->addTopping("Onions");
```

Creational Patterns: Builder

- This can be cumbersome and error-prone
  - We might forget to add green peppers to a Deluxe pizza in one part of our application
- It would be ideal to encapsulate this creation process
- One possible solution involves sub-classing …

Creational Patterns: Builder



Creational Patterns: Builder

- Sub-classing seems like overkill for this application:
  - Our subclasses do not add new state or behaviour
  - Instead, they merely create different representations of the same thing: a pizza!
- How can we create these different representations without adding new sub-classes?

Creational Patterns: Builder

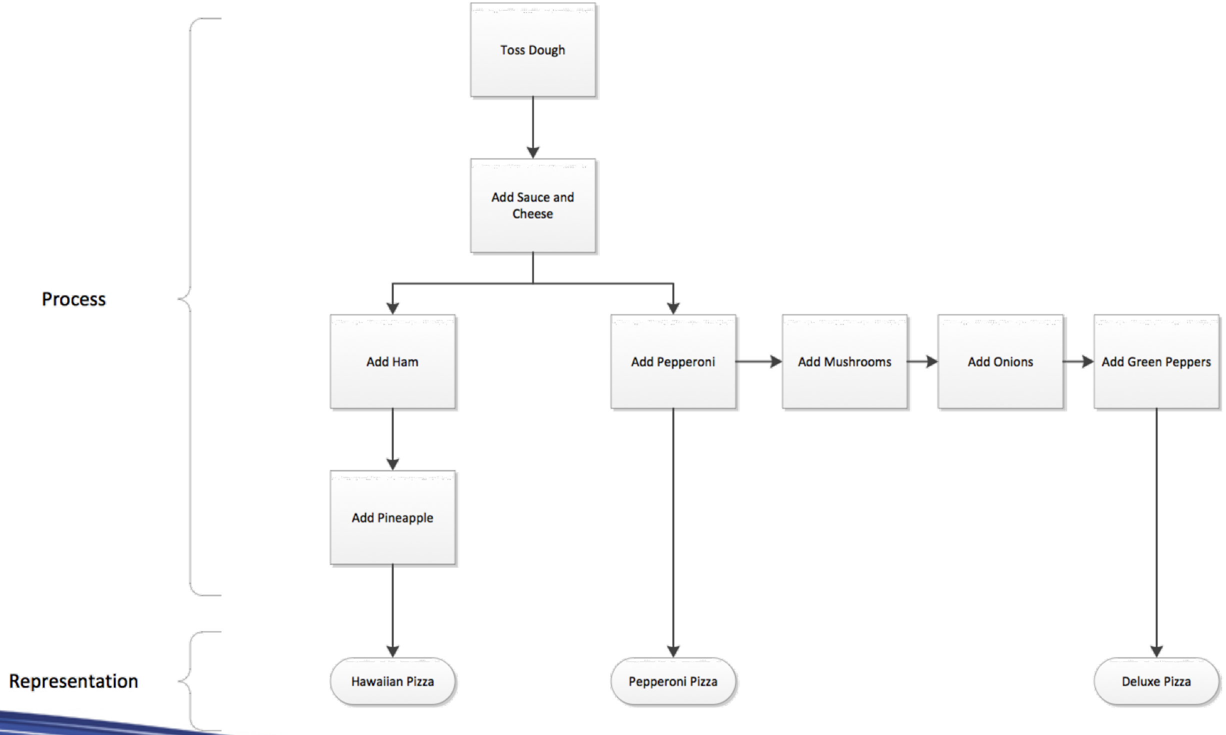
Design Pattern:
Builder

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

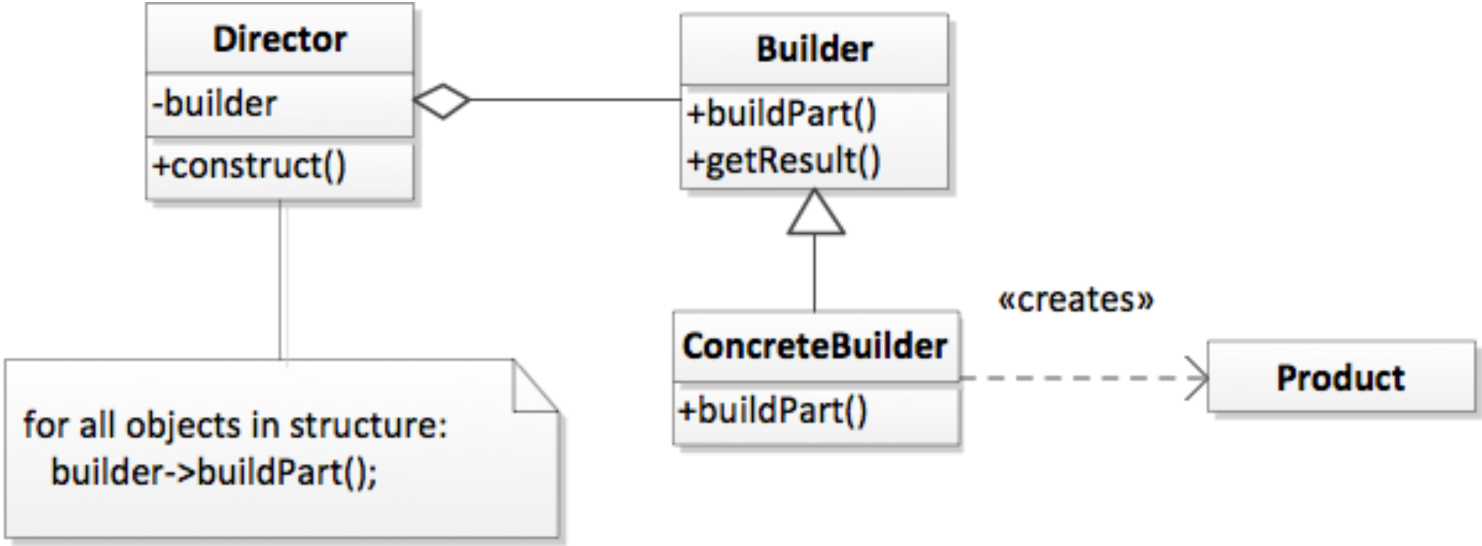
Creational Patterns: Builder

- Applicability:
  - The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled
  - The construction process must allow different representations for the object that's constructed

Creational Patterns: Builder



Creational Patterns: Builder

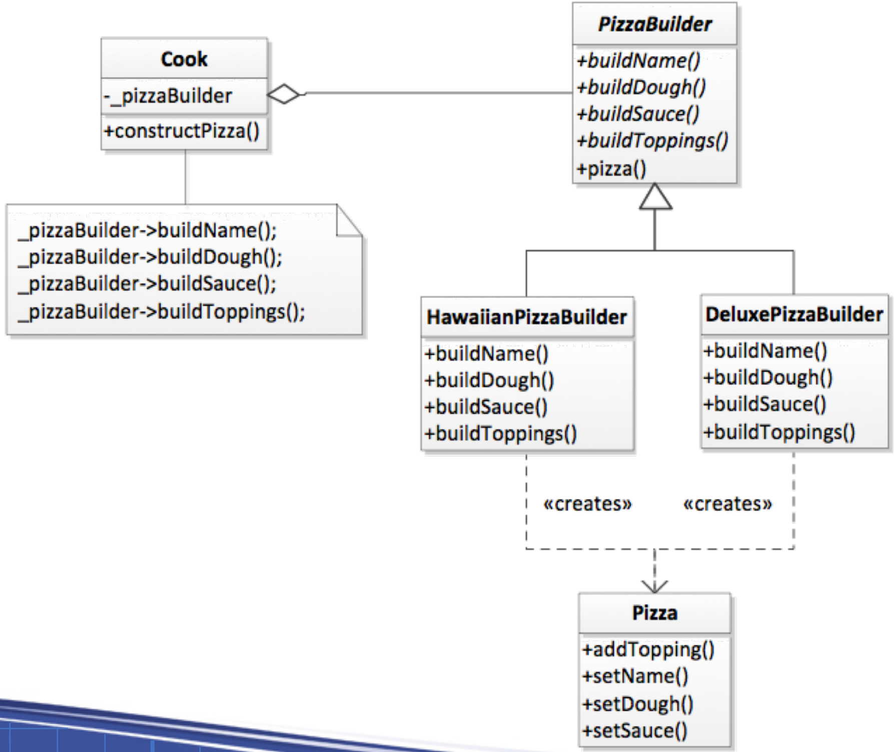


Creational Patterns: Builder

Classes:

- Director
  - Responsible for the sequence of build operations
- Builder
  - Abstract interface for creating products
- Concrete Builder
  - Implements construction and assembly of parts
- Product
  - Object that will be created by Concrete Builder

Creational Patterns: Builder



Creational Patterns: Builder

PizzaBuilder.h

```
// Abstract Builder

class PizzaBuilder
{
   public:
      const Pizza& pizza()
      {
         return _pizza;
      }
      virtual void buildName() = 0;
      virtual void buildDough() = 0;
      virtual void buildSauce() = 0;
      virtual void buildToppings() = 0;

   protected:
    Pizza _pizza;
};
```

Creational Patterns: Builder

HawaiianPizzaBuilder.cpp

```
void HawaiianPizzaBuilder::buildName()
{
   _pizza.setName("Hawaiian");
}

void HawaiianPizzaBuilder::buildDough()
{
   _pizza.setDough("Regular");
}

void HawaiianPizzaBuilder::buildSauce()
{
   _pizza.setSauce("Mild");
}

void HawaiianPizzaBuilder::buildToppings()
{
   _pizza.addTopping("Ham");
   _pizza.addTopping("Pineapple");
}
```

Creational Patterns: Builder

DeluxePizzaBuilder.cpp

```
void DeluxePizzaBuilder::buildName()
{
   _pizza.setName("Deluxe");
}
void DeluxePizzaBuilder::buildDough()
{
   _pizza.setDough("Thick");
}
void DeluxePizzaBuilder::buildSauce()
{
   _pizza.setSauce("Mild");
}
void DeluxePizzaBuilder::buildToppings()
{
   _pizza.addTopping("Pepperoni");
   _pizza.addTopping("Mushrooms");
   _pizza.addTopping("Onions");
   _pizza.addTopping("Green Peppers");
}
```

Creational Patterns: Builder

Cook.cpp

```
Cook::Cook() : _pizzaBuilder(NULL)
{
}
Cook::~Cook()
{
   if (_pizzaBuilder)
      delete _pizzaBuilder;
}
void Cook::setPizzaBuilder(PizzaBuilder* pizzaBuilder)
{
   if (_pizzaBuilder)
      delete _pizzaBuilder;

      _pizzaBuilder = pizzaBuilder;
}
const Pizza& Cook::getPizza()
{
   return _pizzaBuilder->pizza();
}
void Cook::constructPizza()
{
   _pizzaBuilder->buildName();
   _pizzaBuilder->buildDough();
   _pizzaBuilder->buildSauce();
   _pizzaBuilder->buildToppings();
}
```

Creational Patterns: Builder

main.cpp

```
int main()
{

   Cook cook;
   cook.setPizzaBuilder(new HawaiianPizzaBuilder);
   cook.constructPizza();

   Pizza hawaiian = cook.getPizza();
   cout << hawaiian << endl;

   cook.setPizzaBuilder(new DeluxePizzaBuilder);
   cook.constructPizza();

   Pizza deluxe = cook.getPizza();
   cout << deluxe << endl;
}
```

Creational Patterns: Builder

- Consequences:
  - Lets you vary a product’s internal representation
  - Isolates code for construction and representation
  - Gives you finer control over the construction process

Creational Patterns: Builder

- Builder vs. Abstract Factory
  - Abstract Factory
    - Deals with families of related objects
    - Available immediately
  - Builder
    - Creates one, complex product, usually made up of different parts
    - Available via `getResult()`

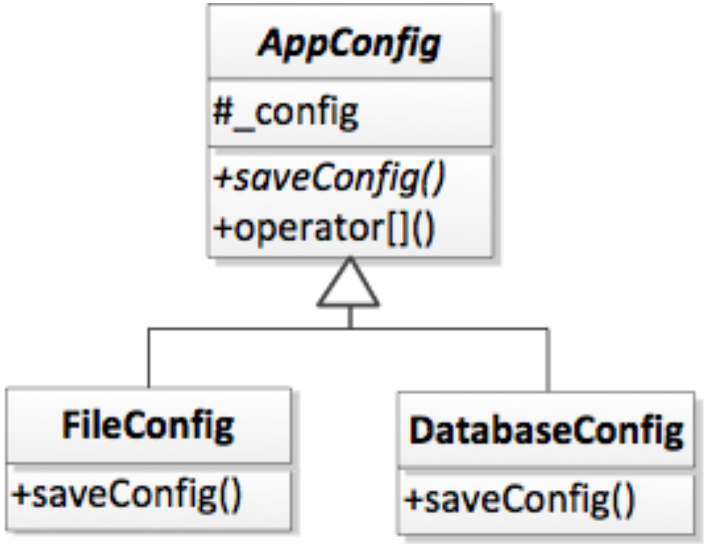
Creational Design Patterns

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

Creational Patterns: Prototype

- Suppose we have a set of classes to load our application configuration from a database, a file, etc.
  - Our configuration is large and takes a while to load
  - Sometimes, we must duplicate our configuration objects
    - e.g. We might want to make changes to one configuration object and save it to a different configuration file without changing the original object

Creational Patterns: Prototype



Creational Patterns: Prototype

AppConfig.h

```
class AppConfig
{
   public:
      virtual void saveConfig() = 0;
      const std::string& operator[](const std::string& key)
      {
         return this->_config[key];
      }

   protected:
      std::map<std::string, std::string> _config;
};
```

Creational Patterns: Prototype

DatabaseConfig.cpp

```
DatabaseConfig::DatabaseConfig(const string& hostname, int port, const string& username,
                               const string& password)
{
   // Simulate load of large configuration data from remote database server
   sleep(3 + (rand() % 3));

   // Simulate adding configuration from the file
   this->_config["config_source"] = hostname;

   // ...
}

void DatabaseConfig::saveConfig()
{
   // ...
}
```

Creational Patterns: Prototype

FileConfig.cpp

```
FileConfig::FileConfig(const string& filename)
{
   // Simulate load of large configuration file on remote network share
   sleep(2 + (rand() % 2));

   // Simulate adding configuration from the file
   this->_config["config_source"] = filename;

   // ...
}

void FileConfig::saveConfig()
{
   // ...
}
```

Creational Patterns: Prototype

- Our data takes a long time to load
  - Maybe the configuration data is large
  - Maybe we're accessing a remote file on a network share or data in a database
- Need to clone it from time to time
- Why can't we simply use the copy constructor?

Creational Patterns: Prototype

```
void f(AppConfig* cfg)
{
   // Clone cfg using copy constructor? Nope … AppConfig is an abstract class, so we can’t
   // use a constructor with it …
   AppConfig cfg2(*cfg);
}

int main()
{
   AppConfig* cfg = new FileConfig("app.conf");
   f(cfg);
}
```

Creational Patterns: Prototype

- Copy constructors won’t work

- Instead, we’ll just create a new object and reload the configuration each time we need a “clone”...

Creational Patterns: Prototype

main.cpp

```
AppConfig* loadConfig()
{
  boost::timer::auto_cpu_timer t;

  cout << "Loading config..." << endl;
  return new FileConfig("/mnt/fileserver/app.conf");
}

int main()
{
  AppConfig* cfg1 = loadConfig();
  AppConfig* cfg2 = loadConfig();
}
```

Creational Patterns: Prototype

```
Output
Loading config...
 3.000832s wall
Loading config...
 3.000379s wall
```

- We take an expensive performance hit each time we reload the configuration
- Can we avoid this somehow?

Creational Patterns: Prototype

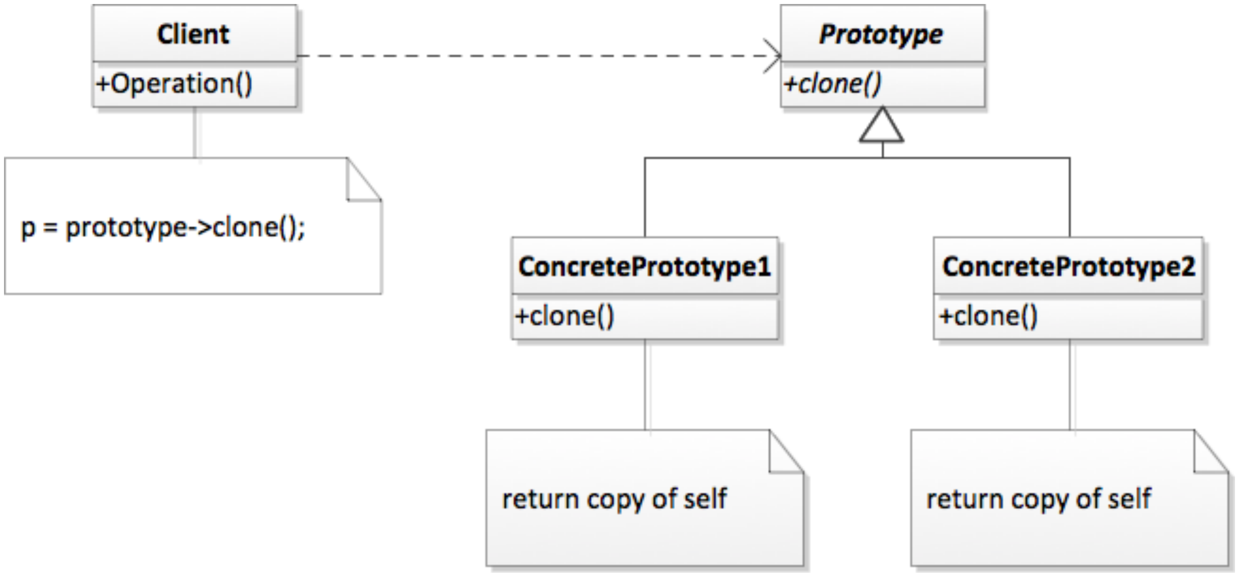
Design Pattern:
Prototype

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying the prototype.

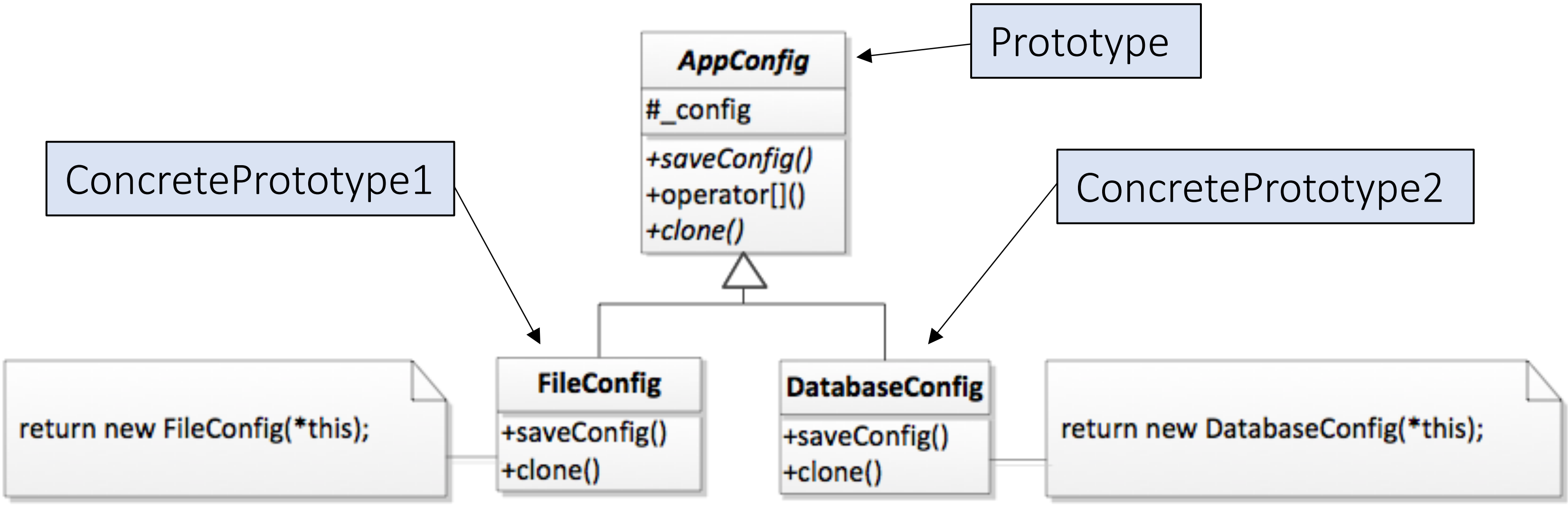
Creational Patterns: Prototype

- Applicability:
  - When the classes to instantiate are specified at run-time, for example, by dynamic loading; or
  - When instances are expensive to create, but easy to copy; or
  - When instances of a class can have one of only a few different combinations of state; in such a case, it may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state

Creational Patterns: Prototype



Creational Patterns: Prototype



Creational Patterns: Prototype

AppConfig.h

```
class AppConfig
{
   public:
      virtual~AppConfig()
      {
      }

      virtual AppConfig* clone() const = 0;
      virtual void saveConfig() = 0;

      const std::string& operator[](const std::string& key)
      {
         return this->_config[key];
      }
   protected:
      std::map<std::string, std::string> _config;
};
```

Creational Patterns: Prototype

DatabaseConfig.cpp

```
AppConfig* DatabaseConfig::clone() const
{
   return new DatabaseConfig(*this);
}
```

Creational Patterns: Prototype

FileConfig.cpp

```
AppConfig* FileConfig::clone() const
{
   return new FileConfig(*this);
}
```

Creational Patterns: Prototype

main.cpp

```
AppConfig* loadConfig()
{
   boost::timer::auto_cpu_timer t;

   cout << "Loading config..." << endl;
   return new FileConfig("/mnt/fileserver/app.conf");
}

int main()
{
   AppConfig* cfg1 = loadConfig();

   boost::timer::auto_cpu_timer t;
   cout << "Cloning config..." << endl;
   AppConfig* cfg2 = cfg1->clone();
}
```

Creational Patterns: Prototype

- Before:

```
Output

Loading config...
 3.000832s wall
Loading config...
 3.000379s wall
```

- After:

```
Output

Loading config...
 3.001179s wall
Cloning config...
 0.000008s wall
```

Creational Patterns: Prototype

- Another example:
  - When creating a game level, we could pass prototypes to use when creating and populating the level

```
GameLevel myLevel(FireMonster, IceSky, GlassWalls, ...)
```

Creational Patterns: Prototype

- Prototype vs Abstract Factory
  - Abstract Factory

```
GameLevel myLevel(FireObjectFactory)
```

    - Creates a family of related products; enforces constraint that they belong together
    - Likely need a factory subclass for each type of level (Fire, Ice, Electric, etc.)
  - Prototype

```
GameLevel myLevel(FireMonster, IceSky, GlassWalls, ...)
```

    - Prototypes allow more flexible mixes of objects
    - May reduce need to have extensive factory hierarchy, especially if there are many different combinations

Creational Patterns: Prototype

- Can use Abstract Factory and Prototype together:

```
Monster* m = new FireMonster();
Wall* w = new IceWall();
Sky* s = new ElectricSky();

ObjectFactory* f = new ObjectFactory(m, w, s);

// ...

// Creates the monster by cloning the
// prototype passed in
Monster* monster = f->createMonster();
```

Creational Patterns: Prototype

- For further flexibility, we could modify our factory to return a random monster from a pool of prototypes:

```
class ObjectFactory
{
   public:
      void addMonsterPrototype(Monster* prototype)
      {
         this->_monsterPrototypes.push_back(prototype);
      }
      Monster* createMonster()
      {
         int idx = random() % this->_monsterPrototypes.size();
         return this->_monsterPrototypes[idx].clone();
      }
   protected:
      std::vector<Monster*> _monsterPrototypes;
};
```

Creational Patterns: Prototype

- Consequences:
  - Hides the concrete product classes from the client – we don’t have to know which concrete type we’re cloning
  - Specify new objects by varying values
  - Configuring an application with classes dynamically
  - Add/remove varieties at run time from a pool of prototypes
  - May reduce need for subclassing
    - Dragons, salamanders, etc. may not have to be subclasses – just generic FireMonsters cloned and then given different characteristics

Creational Patterns: Prototype

- Consequences:
  - May even remove need for Factory subclasses
    - Fire object factory = generic ObjectFactory given several FireMonsters as prototypes
    - Ice object factory = generic ObjectFactory given several IceMonsters as prototypes

Creational Design Patterns

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