Object Oriented Software Design II

Real Application Design

Christian Nastasi
http://retis.sssup.it/~lipari
http://retis.sssup.it/~chris/cpp

Scuola Superiore Sant'Anna – Pisa

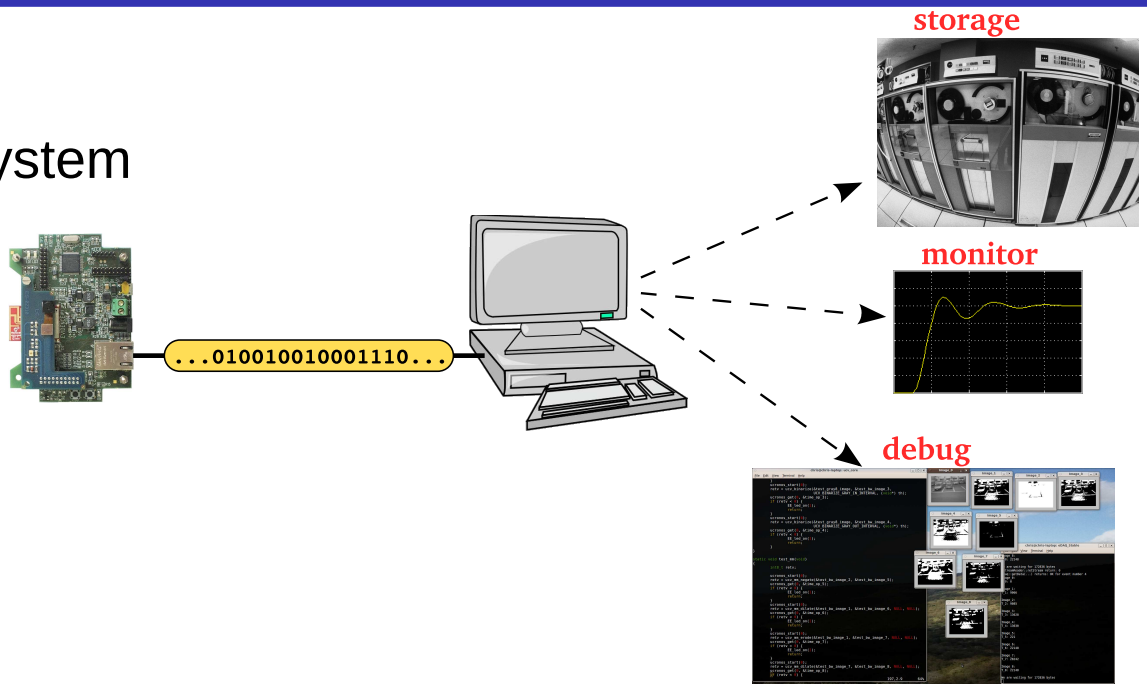
April 12, 2012

Outline

Outline

Data AcQuisition System

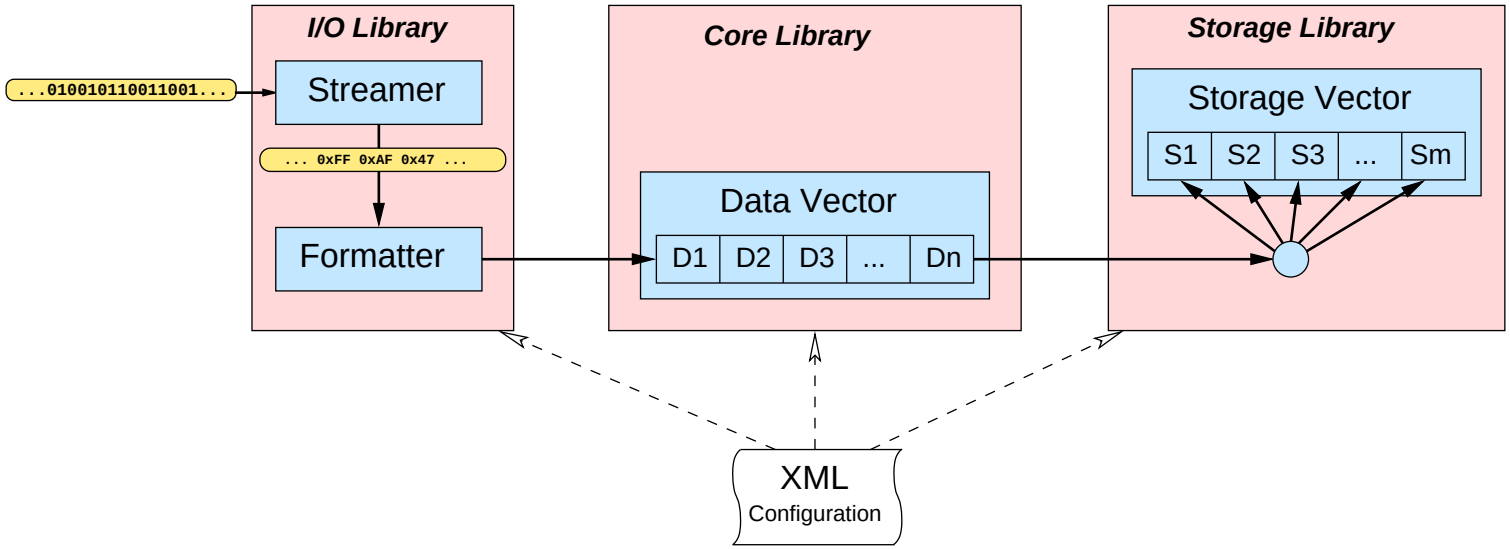
A basic DAQ system



duck-lab: an open-source DAQ framework

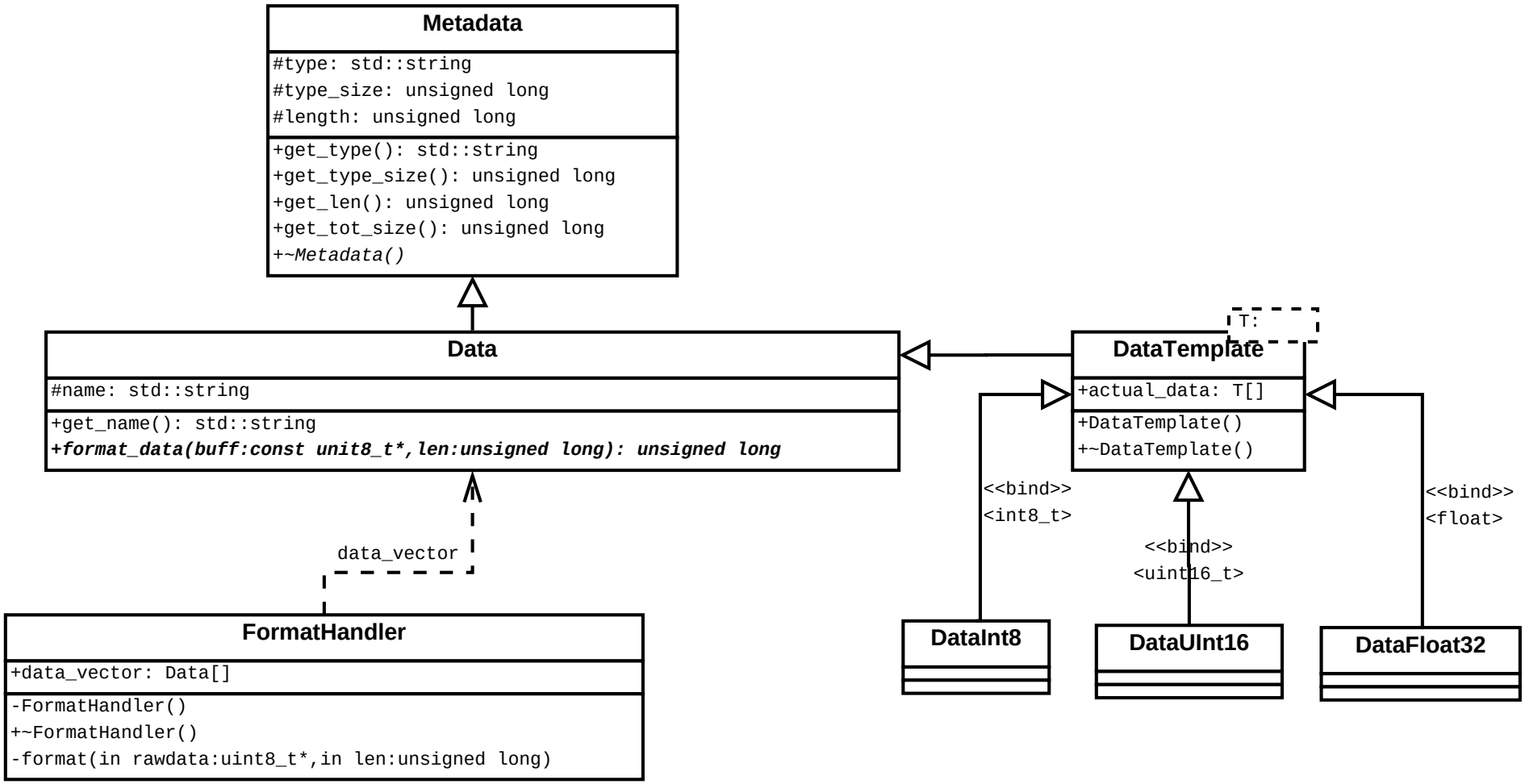
- Project (temporary) home page:
  `http://retis.sssup.it/~chris/duck-lab`
- Mercurial revision control: `https://rtn.sssup.it/hg/duck-lab`
  `(hg clone https://rtn.sssup.it/hg/duck-lab)`

duck-lab architecture



- We have defined a hierarchy for `Data`
- We have defined the interaction with the Formatter, implemented by the `FormatHandler` class

duck-lab architecture



The `FormatHandler` class

- We have done `Data` creation/destruction
- We have done `Data` formatting

```
class FormatHandler {
public:
    FormatHandler();
    ~FormatHandler();
    void format(const uint8_t* rawdata,
                unsigned long len);
private:
    std::vector<Data*> data_vector;
};
```

The `FormatHandler` class

- We have done `Data` creation/destruction
- We have done `Data` formatting

```
class FormatHandler {
public:
    FormatHandler();
    ~FormatHandler();
    void format(const uint8_t* rawdata,
                unsigned long len);
private:
    std::vector<Data*> data_vector;
};
```

Switch-based concrete object allocation (bad)

Polymorphic destruction (good!)

Polymorphic formatting with delegation to the `Data` classes (good!)

The `FormatHandler` class

- We have done `Data` creation/destruction
- We have done `Data` formatting

```
class FormatHandler {
public:
    FormatHandler();
    ~FormatHandler();
    void format(const uint8_t* rawdata,
                unsigned long len);
private:
    std::vector<Data*> data_vector;
};
```

Switch-based concrete object allocation (bad)

Polymorphic destruction (good!)

Polymorphic formatting with delegation to the `Data` classes (good!)

- So, given the following configuration (XML)

```
<message_description>
   <data data_type="uint16"  data_name="data_name_1" data_len="2"/>
   <data data_type="float32" data_name="data_name_2" data_len="1"/>
   <data data_type="int8"    data_name="data_name_3" data_len="4"/>
</message_description>
```

FormatHandler::FormatHandler()

DataExample3.cc

```
FormatHandler::FormatHandler()
{
        // We create the data vector according to XML config
        vector<XmlDataElement> config = parse_xml_data();

        vector<XmlDataElement>::const_iterator it;
        for (it = config.begin(); it != config.end(); ++it) {
                Data *d;
                if (it->type == "int8")
                        d = new DataInt8(it->name, it->len);
                else if (it->type == "uint16")
                        d = new DataUInt16(it->name, it->len);
                else if (it->type == "float32")
                        d = new DataFloat32(it->name, it->len);
                else {
                        cout << "Unknown data type = '" << it->type <<
                                "' ! Skipping" << endl;
                        continue;
                }
                data_vector.push_back(d);
        }
}
```

- **Unmanageable**: adding new new data requires adding `else if`.

FormatHandler::~FormatHandler()

DataExample3.cc

```
FormatHandler::~FormatHandler()
{
        // We have to destroy the data vector properly
        vector<Data*>::iterator it;
        for (it = data_vector.begin(); it != data_vector.end(); ++it)
                delete *it;
}
```

FormatHandler::format()

DataExample3.cc

```
void FormatHandler::format(uint8_t* rawdata, unsigned long len)
{
        vector<Data*>::iterator it;
        for (it = data_vector.begin(); it != data_vector.end(); ++it) {
                unsigned long required_len = (*it)->get_tot_size();
                if (required_len > len) {
                        cout << "Not enough bytes to read Data. Returning" <<
                                endl;
                        return;
                }
                if ((*it)->get_type() == "int8") {
                        DataInt8 *d = static_cast<DataInt8*>(*it);
                        // memcpy if fine with int8
                        memcpy(d->actual_data, rawdata, required_len);
                } else if ((*it)->get_type() == "uint16") {
                        DataUInt16 *d = static_cast<DataUInt16*>(*it);
                        // Fix endianess before memcpy!
                        memcpy(d->actual_data, rawdata, required_len);
                } else if ((*it)->get_type() == "float32") {
                        DataFloat32 *d = static_cast<DataFloat32*>(*it);
                        // Fix endianess before memcpy!
                        memcpy(d->actual_data, rawdata, required_len);
                } else {
                        cout << "Unknown data type = '" << (*it)->get_type() <<
                                "' ! Skipping" << endl;
                        continue;
                }
                len -= required_len;
                rawdata += required_len;
        }
}
```

FormatHandler::format()

```
class Data : public Metadata {
//...
public:
    virtual unsigned long format_data(const uint8_t* buff, unsigned long len) = 0;
//...
};
```

DataExample4.cc

```
void FormatHandler::format(uint8_t* rawdata, unsigned long len)
{
        vector<Data*>::iterator it;
        for (it = data_vector.begin(); it != data_vector.end(); ++it) {
                unsigned long used_bytes = (*it)->format_data(rawdata, len);
                len -= used_bytes;
                rawdata += used_bytes;
        }
}
```

```
template<class T>
class DataTemplate : public Data {
//...
public:
    unsigned long format_data(const uint8_t* buff, unsigned long len)
    {
        unsigned long required_len = get_tot_size();
        if (required_len > len) {
            std::cout << "Not enough bytes to read Data." << std::endl;
            return 0;
        }
        memcpy(actual_data, buff, required_len);
        return required_len;
    };
//...
};
```

What’s next

- We’ll see how to solve the Data Vector creation problem

What's next

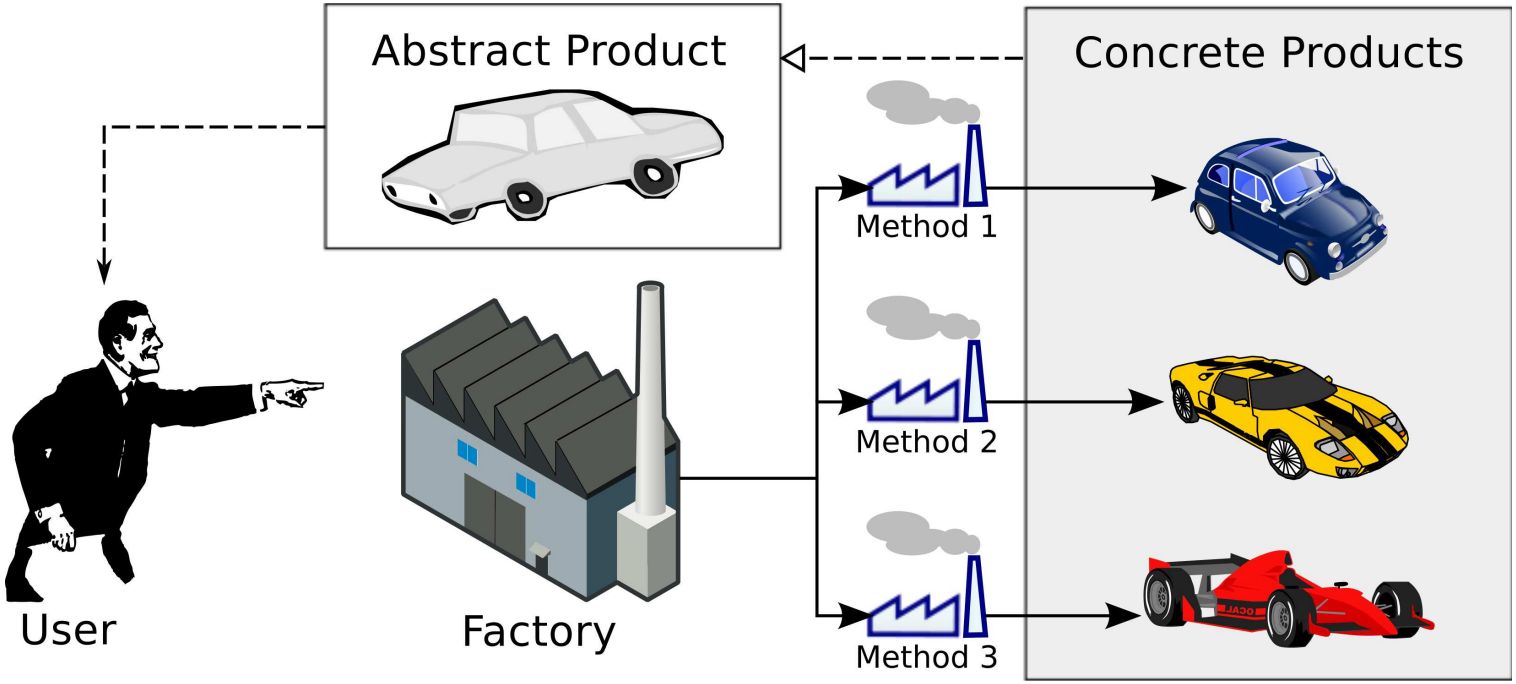
- We'll see how to solve the Data Vector creation problem
- Using delegation in `FormatHandler::FormatHandler()`
- Using a design pattern: *Factory Method*

Outline

Outline

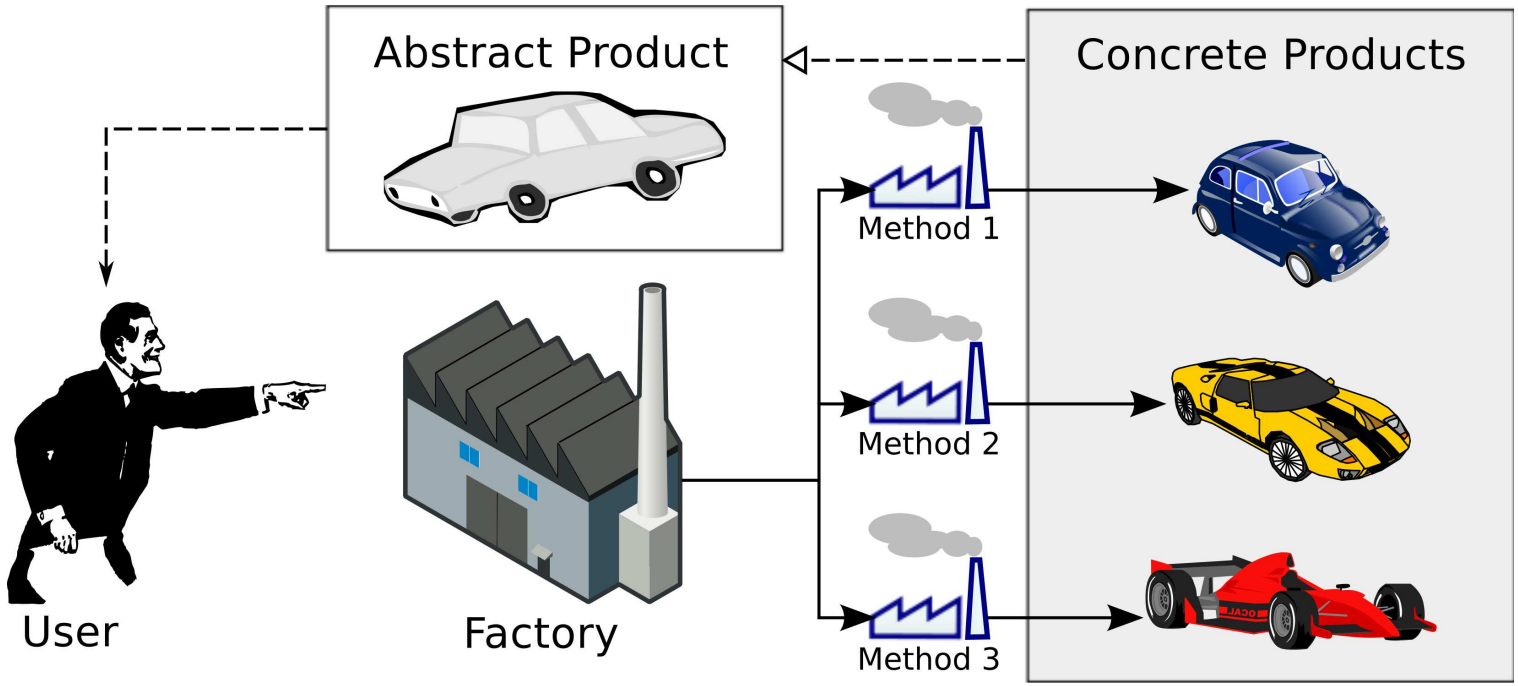
Class Factory

Creational design pattern known as *Factory Method Pattern*



Class Factory

Creational design pattern known as *Factory Method Pattern*

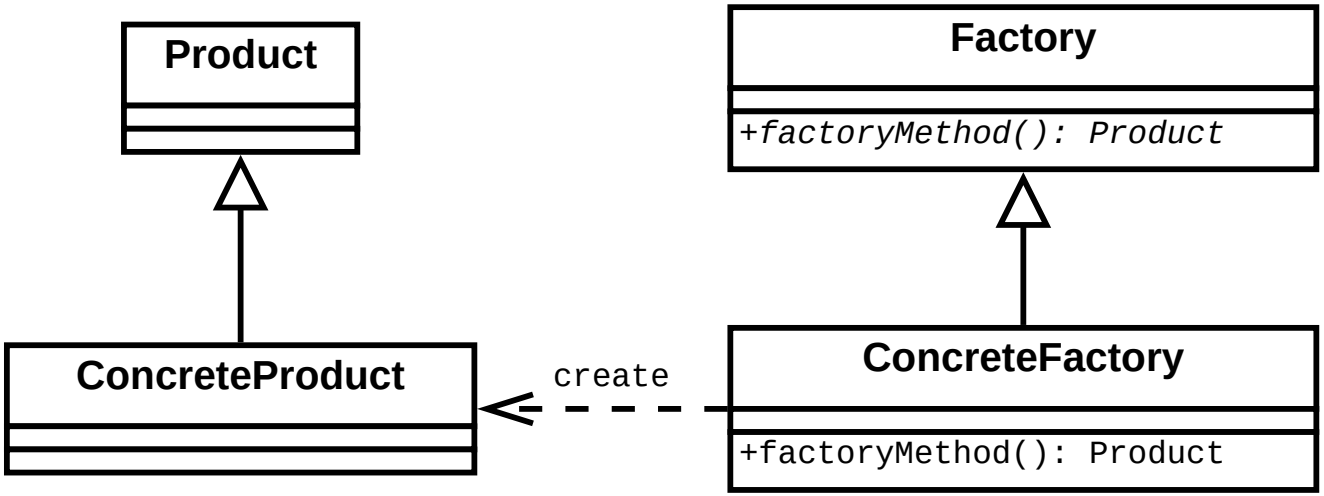


- A *factory* is the location or a concrete class in the code at which objects are constructed.
- The intent in employing the pattern is to insulate the creation of objects from their usage.
- New derived types can be introduced with no change to the code that uses the base class.

[wikipedia]

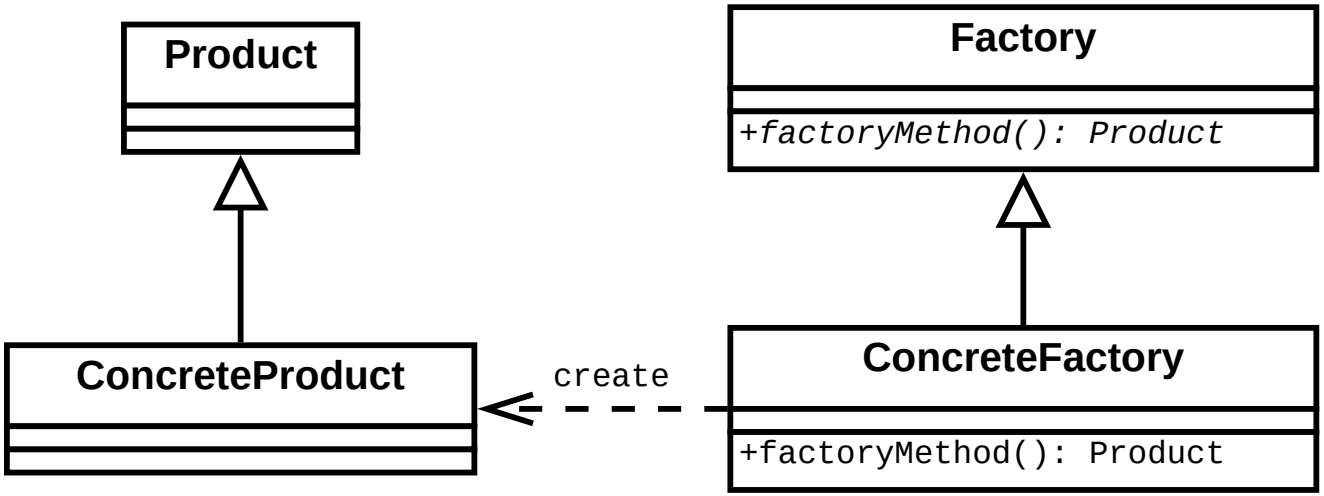
UML class diagram

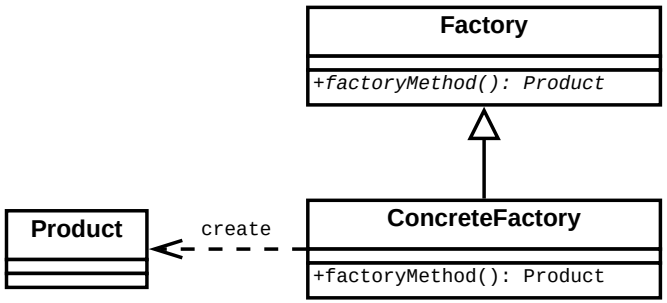
The Factory Method in UML



UML class diagram

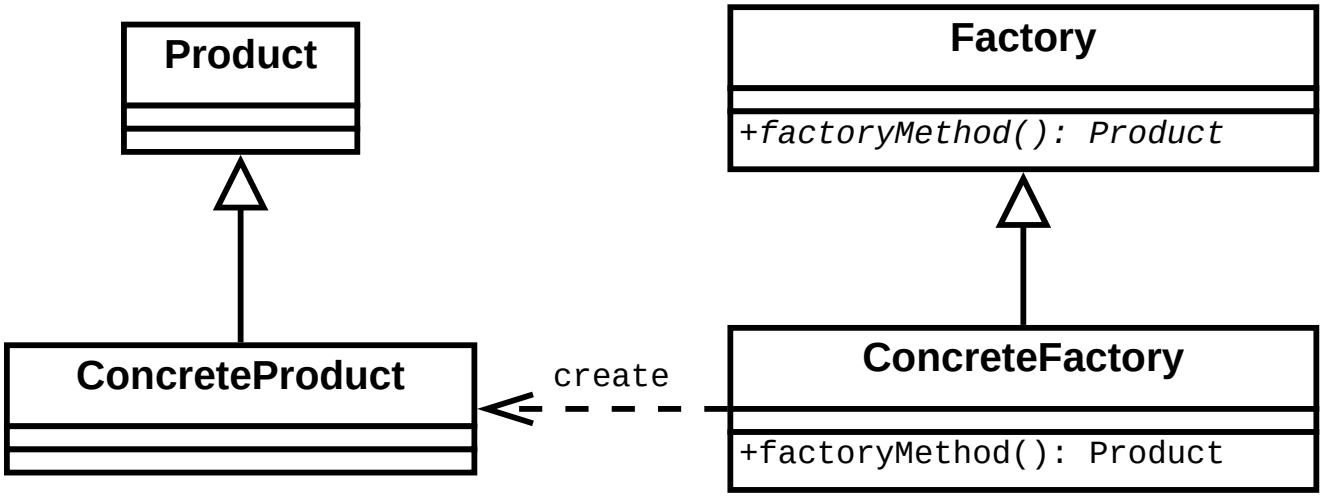
The Factory Method in UML

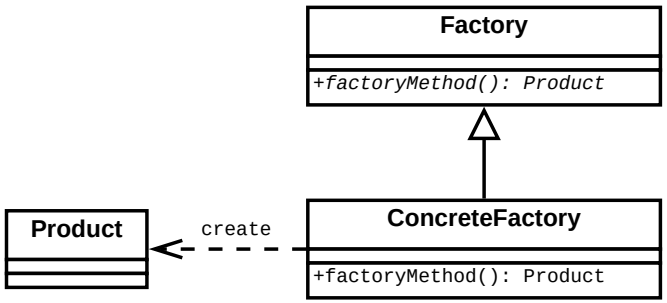


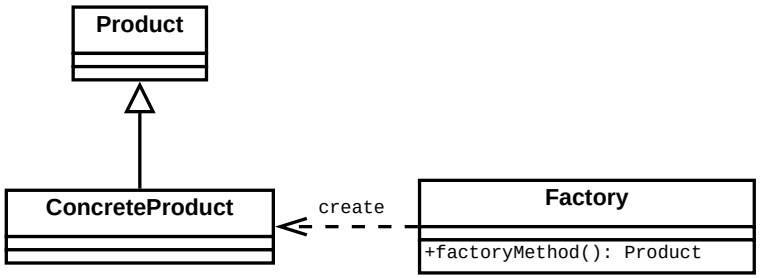


UML class diagram

The Factory Method in UML

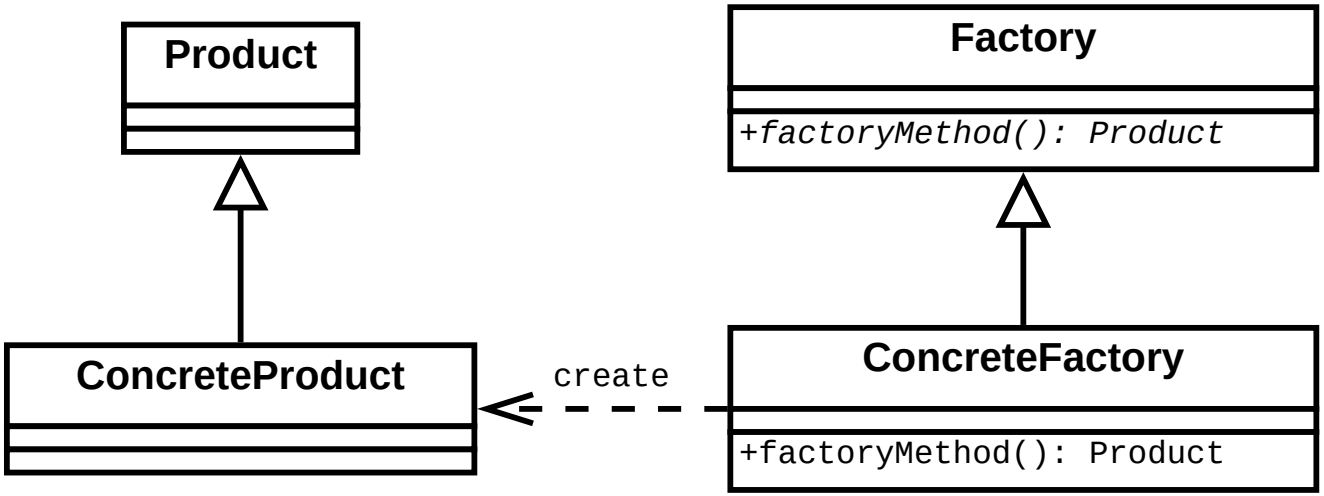


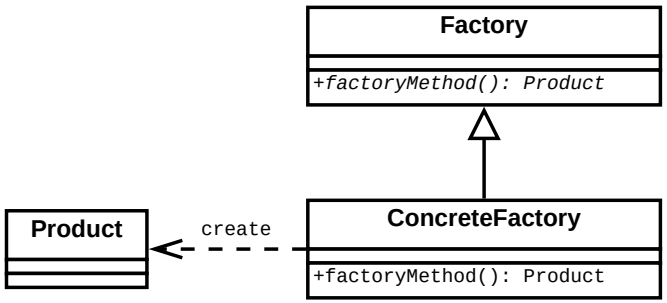


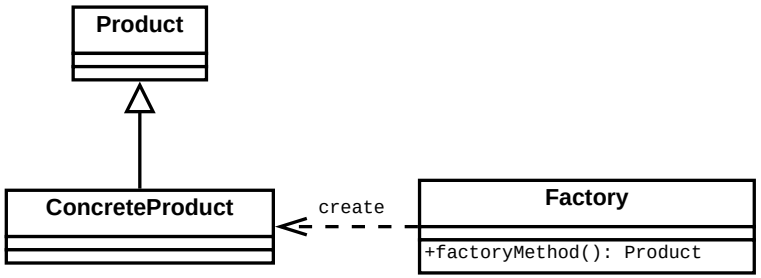


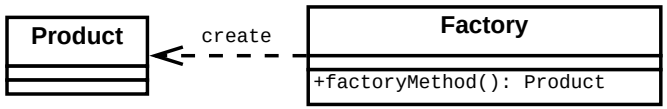
UML class diagram

The Factory Method in UML





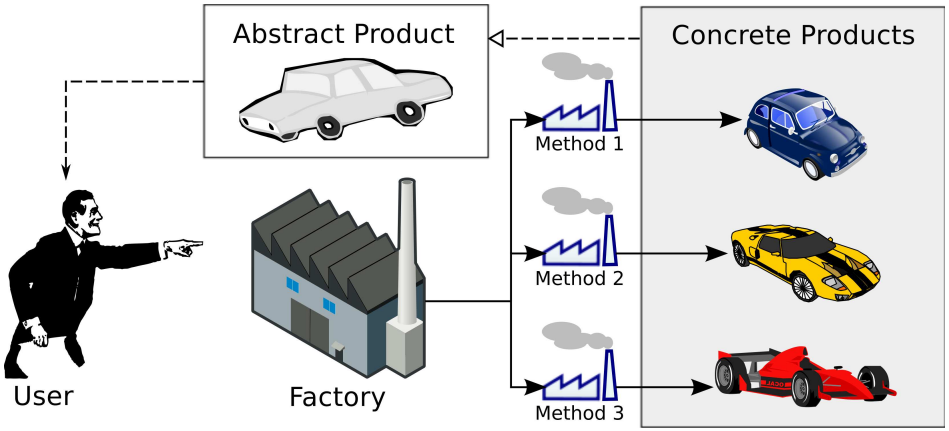




Factory Method Pattern

“Define an interface for creating an object, but let the classes which implement the interface decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses.”

[Design Patterns: Elements of Reusable Object-Oriented Software]

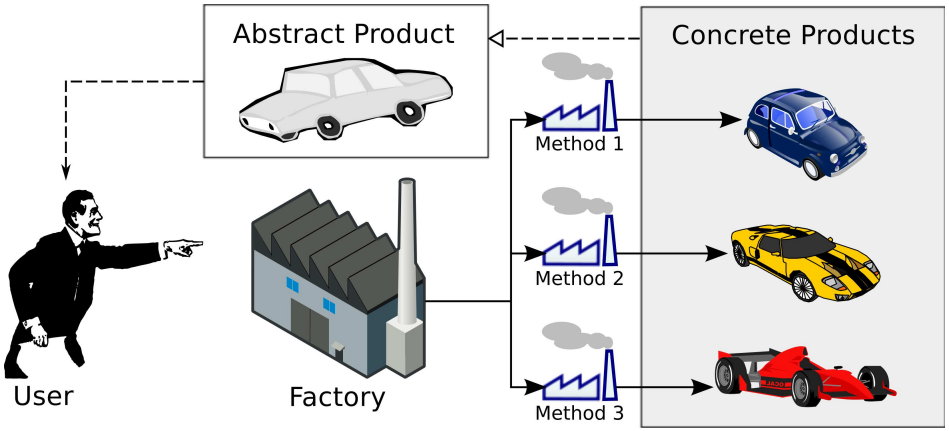


Main benefit?

Factory Method Pattern

“Define an interface for creating an object, but let the classes which implement the interface decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses.”

[Design Patterns: Elements of Reusable Object-Oriented Software]



Main benefit?

- **Delegation**: the *user* does not have access to creation information.

Other benefits

Other benefits

- Named constructor idiom: factory methods rather than constructors

```
class Complex {
     Complex(double a, double b) { /*...*/ };
public:
     static Complex* fromCartesian(double r, double i) { return new Complex(r, i); }

     static Complex* fromPolar(double m, double a)
                                        {return new Complex(m * cos(a), m * sin(a)); }
}
Complex* c1 = Complex.fromPolar(1, 3.14);     // creates z = -1
Complex* c2 = Complex.fromCartesian(1, 3.14); // creates z = 1 + j3.14
```

Other benefits

- Named constructor idiom: factory methods rather than constructors

```
class Complex {
     Complex(double a, double b) { /*...*/ };
public:
     static Complex* fromCartesian(double r, double i) { return new Complex(r, i); }

     static Complex* fromPolar(double m, double a)
                                        {return new Complex(m * cos(a), m * sin(a)); }
}
Complex* c1 = Complex.fromPolar(1, 3.14);     // creates z = -1
Complex* c2 = Complex.fromCartesian(1, 3.14); // creates z = 1 + j3.14
```

- Encapsulating creation: factory hides creation logic

```
class ClassFactory {
public:
    static Product* create(const std::string& file_name)
    {
        std::string type = get_type_from_file(file_name);
        if (type == "type_1") {
            new ConcreteProduct1(/*might take params*/);
        } else if (type == "type_2") {
            new ConcreteProduct2(/*might take other params*/);
        } // etc...
    }
}
Product* p = ClassFactory.create("config.xml");
```

Outline

Family of factories

- Sometimes factories might belong to the same family
- Example: the family of *vehicles*
  - *Cars*
  - *Motorcycles*
  - ...

Family of factories

- Sometimes factories might belong to the same family
- Example: the family of *vehicles*
  - *Cars*
  - *Motorcycles*
  - ...

The abstraction of set of factories belonging to the same family is called *Abstract Factory*.

- It's a creational design pattern
- "Provide an interface for creating families of related or dependent objects without specifying their concrete classes"

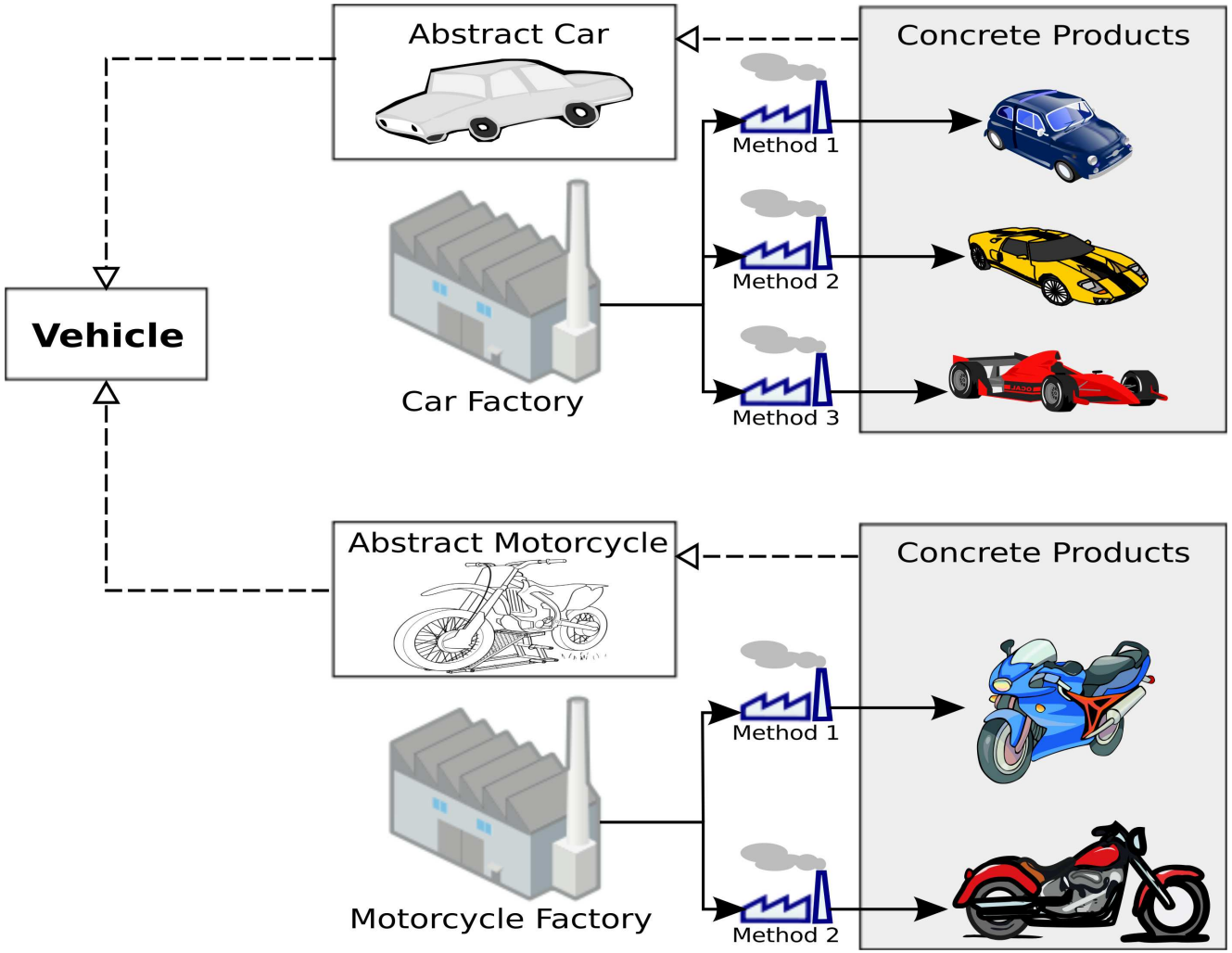
[Design Patterns: Elements of Reusable Object-Oriented Software]

The Abstract Factory Pattern

Given a set of related factories...

The Abstract Factory Pattern

Given a set of related factories...

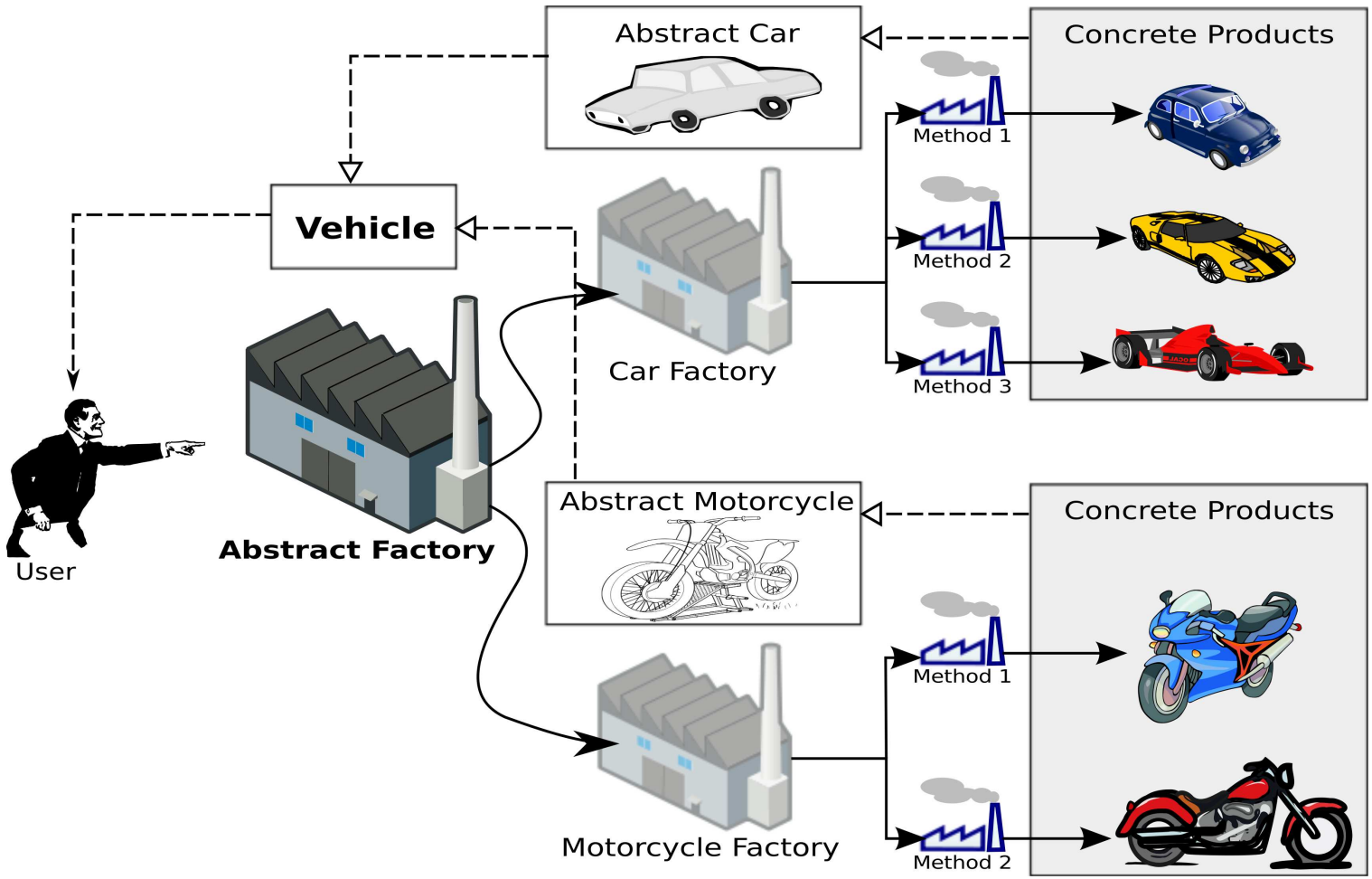


The Abstract Factory Pattern

...create and abstraction (family) of them.

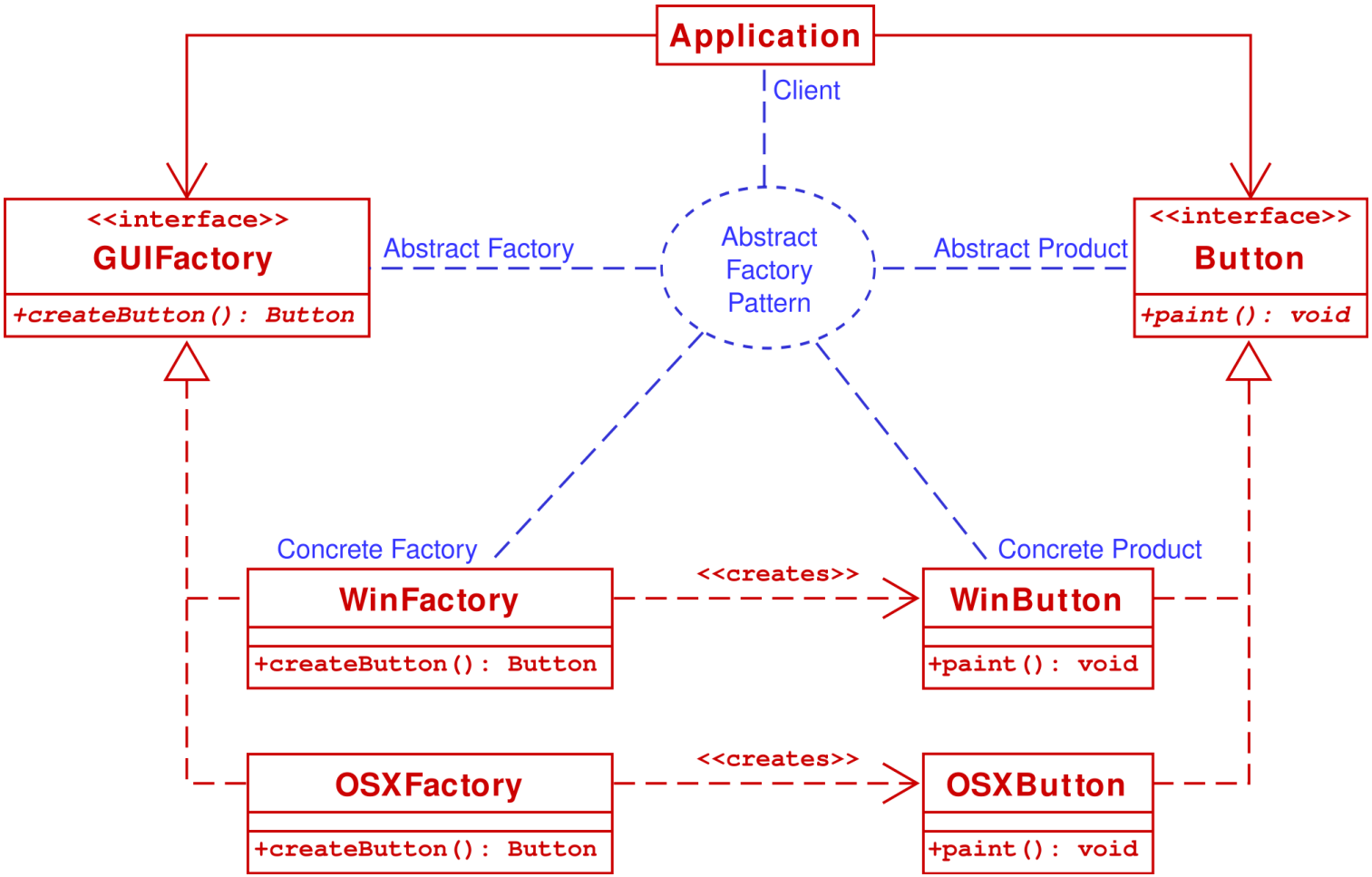
The Abstract Factory Pattern

...create and abstraction (family) of them.



UML class diagram

The Abstract Factory in UML



[wikipedia]

Outline

Example

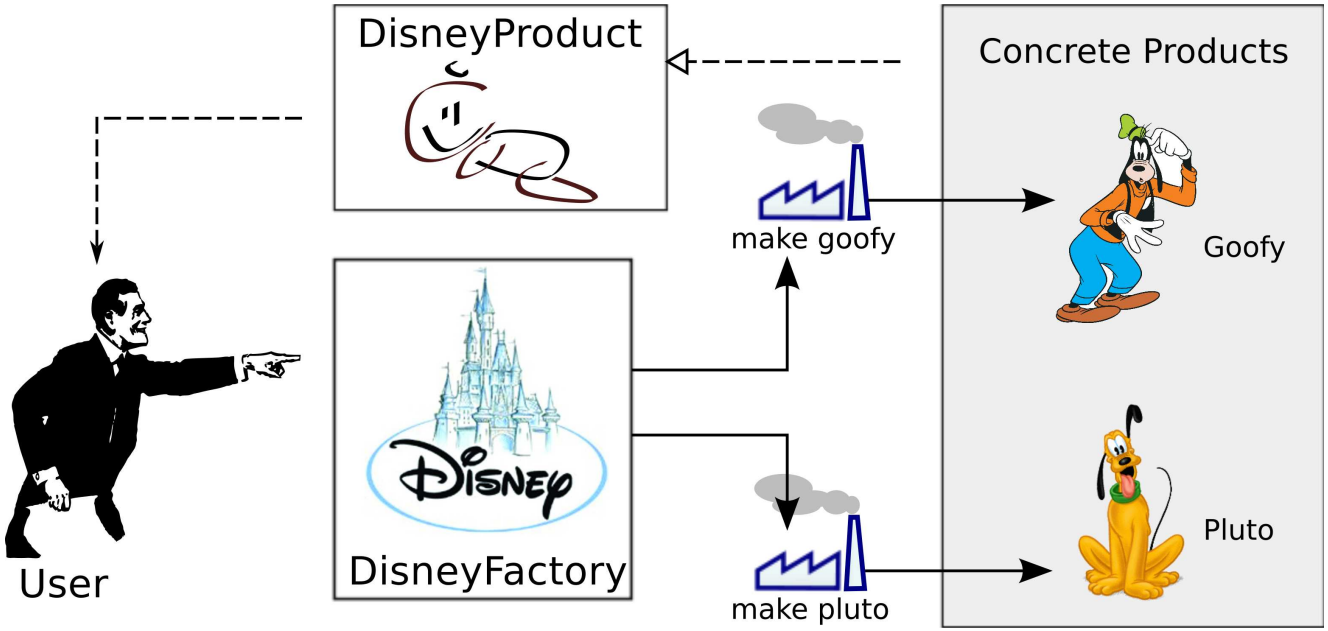
Let's consider the *Disney's cartoons* problem:

- The *user* wants to animate cartoons
- The *user* does not care about their creation process

Example

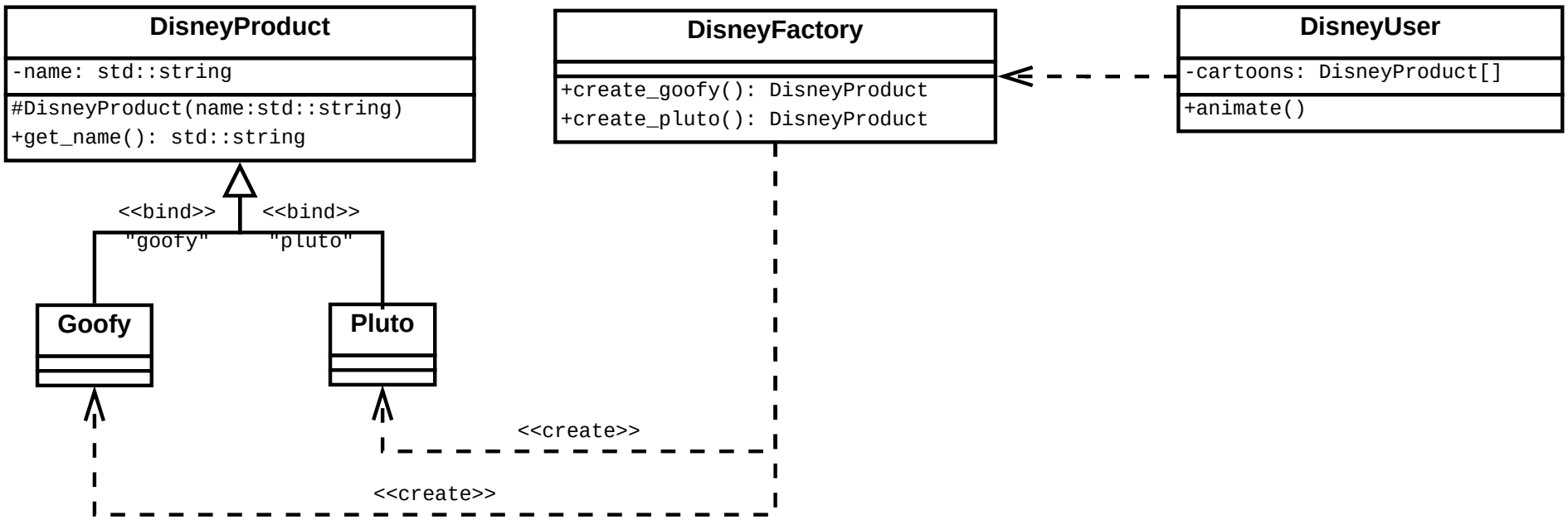
Let's consider the *Disney's cartoons* problem:

- The *user* wants to animate cartoons
- The *user* does not care about their creation process



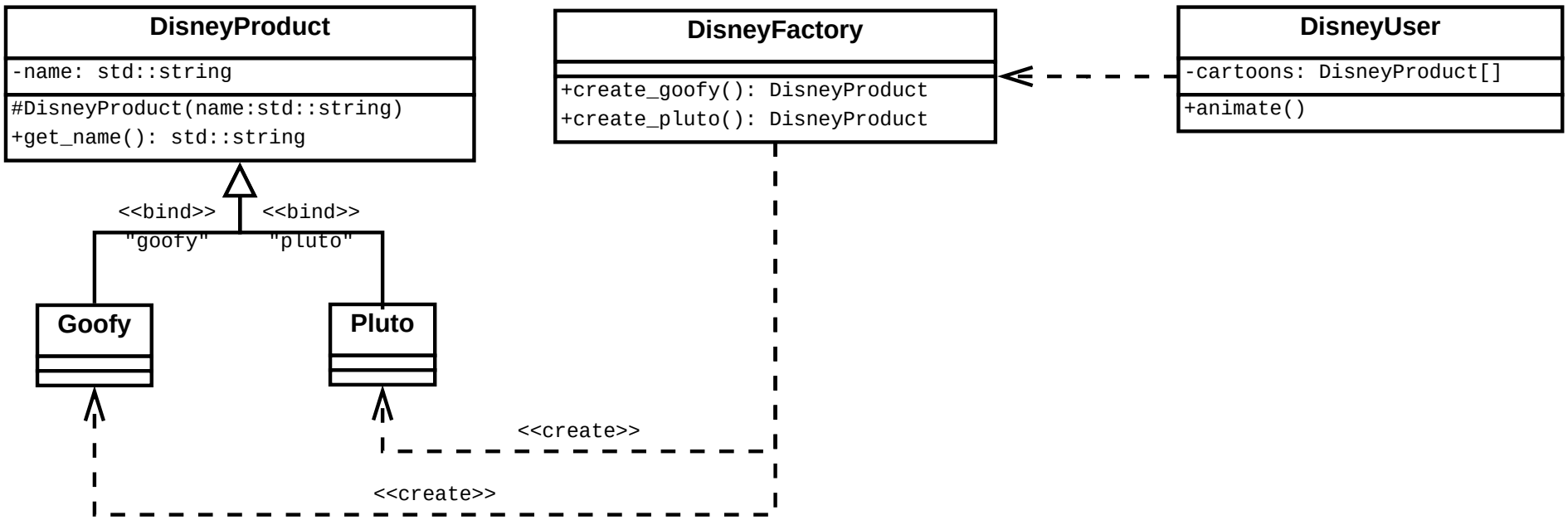
The Disney's cartoons problem

A possible solution (UML class diagram)



The Disney's cartoons problem

A possible solution (UML class diagram)



- Let's try to implement in C++
- Some requirements:
  - The creation of object is exclusively managed by the Factory
  - Delegating as much as we can (i.e. hiding to the User)

C++: DisneyProduct

FactoryExample1.h

```
class DisneyProduct {
public:
        virtual ~DisneyProduct() {};
        inline std::string get_name(void) const { return name; };
protected:
        DisneyProduct(const std::string& n) : name(n) {};
private:
        DisneyProduct(const DisneyProduct&);
        DisneyProduct &operator=(const DisneyProduct&);
        std::string name;
};
```

C++: DisneyProduct

FactoryExample1.h

```
class DisneyProduct {
public:
        virtual ~DisneyProduct() {};
        inline std::string get_name(void) const { return name; };
protected:
        DisneyProduct(const std::string& n) : name(n) {};
private:
        DisneyProduct(const DisneyProduct&);
        DisneyProduct &operator=(const DisneyProduct&);
        std::string name;
};
```

- protected constructor: only derived class can call it

C++: DisneyProduct

FactoryExample1.h

```
class DisneyProduct {
public:
        virtual ~DisneyProduct() {};
        inline std::string get_name(void) const { return name; };
protected:
        DisneyProduct(const std::string& n) : name(n) {};
private:
        DisneyProduct(const DisneyProduct&);
        DisneyProduct &operator=(const DisneyProduct&);
        std::string name;
};
```

- protected constructor: only derived class can call it
- why virtual destructor?

C++: DisneyProduct

FactoryExample1.h

```
class DisneyProduct {
public:
        virtual ~DisneyProduct() {};
        inline std::string get_name(void) const { return name; };
protected:
        DisneyProduct(const std::string& n) : name(n) {};
private:
        DisneyProduct(const DisneyProduct&);
        DisneyProduct &operator=(const DisneyProduct&);
        std::string name;
};
```

- protected constructor: only derived class can call it
- why virtual destructor?
- why other private members?

C++: Goofy, Pluto and DisneyFactory

```
class Goofy : public DisneyProduct {
    Goofy() : DisneyProduct("goofy") {};
};

class Pluto : public DisneyProduct {
    Pluto() : DisneyProduct("pluto") {};
};
```

```
class DisneyFactory {
public:
    DisneyProduct* create_goofy(void) { return new Goofy(); }
    DisneyProduct* create_pluto(void) { return new Pluto(); }
};
```

C++: Goofy, Pluto and DisneyFactory

```
class Goofy : public DisneyProduct {
    Goofy() : DisneyProduct("goofy") {};
};

class Pluto : public DisneyProduct {
    Pluto() : DisneyProduct("pluto") {};
};
```

```
class DisneyFactory {
public:
    DisneyProduct* create_goofy(void) { return new Goofy(); }
    DisneyProduct* create_pluto(void) { return new Pluto(); }
};
```

- **Problem**: does not compile, why?

C++: Goofy, Pluto and DisneyFactory

```
class Goofy : public DisneyProduct {
    Goofy() : DisneyProduct("goofy") {};
};

class Pluto : public DisneyProduct {
    Pluto() : DisneyProduct("pluto") {};
};
```

```
class DisneyFactory {
public:
    DisneyProduct* create_goofy(void) { return new Goofy(); }
    DisneyProduct* create_pluto(void) { return new Pluto(); }
};
```

- **Problem**: does not compile, why?
- `Goofy::Goofy()` and `Pluto::Pluto()` are private, hidden to `DisneyFactory`.

C++: Goofy, Pluto and DisneyFactory

```
class Goofy : public DisneyProduct {
    Goofy() : DisneyProduct("goofy") {};
};

class Pluto : public DisneyProduct {
    Pluto() : DisneyProduct("pluto") {};
};
```

```
class DisneyFactory {
public:
    DisneyProduct* create_goofy(void) { return new Goofy(); }
    DisneyProduct* create_pluto(void) { return new Pluto(); }
};
```

- **Problem**: does not compile, why?
- `Goofy::Goofy()` and `Pluto::Pluto()` are private, hidden to `DisneyFactory`.
- **Solution**: using *friendship*

C++: friendly Goofy and Pluto

After all, Goofy and Pluto are Disney's friends :)

FactoryExample1.h

```
class Goofy : public DisneyProduct {
private:
        Goofy() : DisneyProduct("goofy") {};
        friend class DisneyFactory;
};

class Pluto : public DisneyProduct {
private:
        Pluto() : DisneyProduct("pluto") {};
        friend class DisneyFactory;
};
```

This is now valid!

```
class DisneyFactory {
public:
    DisneyProduct* create_goofy(void) { return new Goofy(); }
    DisneyProduct* create_pluto(void) { return new Pluto(); }
};
```

C++: the DisneyUser

FactoryExample1.h

```
enum DisneyCharacterID {
        GOOFY_ID = 0,
        PLUTO_ID = 1,
};
```

FactoryExample1.h

```
class DisneyUser {
public:
        DisneyUser();
        ~DisneyUser();
        void animate(void);
private:
        DisneyProduct* create_cartoon(int id);

        DisneyProduct** toons;
        size_t len_toons;
};
```

```
DisneyUser::DisneyUser()
{
    const int config[] = {GOOFY_ID, PLUTO_ID,
                          PLUTO_ID, 1234/*invalid*/};

    len_toons = 4;
    toons = new DisneyProduct*[4];
    for (size_t i = 0; i < 4; i++)
        toons[i] = create_cartoon(config[i]);
}

DisneyUser::~DisneyUser()
{
    for (size_t i = 0; i < len_toons; i++)
        if (toons[i])
            delete toons[i];
    delete[] toons;
}
```

Equivalent to a configuration file

Allocation of the cartoon container: array of `DisneyProduct*`

Delete each cartoon from the container

Delete the container itself

C++: the DisneyUser

FactoryExample1.cc

```
DisneyProduct* DisneyUser::create_cartoon(int id)
{
        DisneyFactory factory;
        switch(id) {
        case GOOFY_ID:
                return factory.create_goofy();
        case PLUTO_ID:
                return factory.create_pluto();
        }
        return NULL;
}

void DisneyUser::animate(void)
{
        for (size_t i = 0; i < len_toons; i++)
                if (toons[i])
                        cout << "Hullo, my name is " <<
                                toons[i]->get_name() << endl;
}
```

The full example is:

`./examples/09.real-application_factories-examples/FactoryExample1.c`

First improvement

Of course we don’t like the first implementation...

First improvement

Of course we don’t like the first implementation...

- The `DisneyProduct*` container is so old-fashioned

```
class DisneyUser {
private:
    DisneyProduct** toons;
    size_t len_toons;
//...
};
```

First improvement

Of course we don't like the first implementation...

- The `DisneyProduct*` container is so old-fashioned

```
class DisneyUser {
private:
    DisneyProduct** toons;
    size_t len_toons;
//...
};
```

- Part of the creation logic is on the *User* side

```
DisneyProduct* DisneyUser::create_cartoon(int id)
{
    DisneyFactory factory;
    switch(id) {
    case GOOFY_ID:
        return factory.create_goofy();
    case PLUTO_ID:
        return factory.create_pluto();
    }
    return NULL;
}
<<<<<<< HEAD
```

First improvement

Solution:

First improvement

Solution:

- Let’s use an STL container...

First improvement

Solution:

- Let's use an STL container... `std::vector`

```
class DisneyUser {
    std::vector<DisneyProduct*> toons;
//...
};
```

First improvement

Solution:

- Let's use an STL container... `std::vector`

```
class DisneyUser {
    std::vector<DisneyProduct*> toons;
//...
};
```

- Let's encapsulate the `create_cartoon()` in the Factory

FactoryExample2.cc

```
DisneyProduct* DisneyFactory::create(int id)
{
        switch(id) {
        case GOOFY_ID:
                return new Goofy();
        case PLUTO_ID:
                return new Pluto();
        }
        return NULL;
}
```

FactoryExample2.cc

```
DisneyUser::DisneyUser()
{
        const int config[] = {GOOFY_ID, PLUTO_ID,
                              PLUTO_ID, 1234/*invalid*/};

        DisneyFactory factory;

        for (size_t i = 0; i < 4; i++) {
                DisneyProduct *p = factory.create(config[i]);
                if (p)
                        toons.push_back(p);
        }
}
```

The full example is:

`./examples/09.real-application_factories-examples/FactoryExample2.c`

Second improvement

Can we get rid of the `switch`-based implementation?

Second improvement

Can we get rid of the `switch`-based implementation?

- Use a look-up table

ID	Factory
GOOFY_ID	Goofy Creator
PLUTO_ID	Pluto Creator
...	...

Second improvement

Can we get rid of the `switch`-based implementation?

- Use a look-up table

ID	Factory
GOOFY_ID	Goofy Creator
PLUTO_ID	Pluto Creator
...	...

- **Q**: What shall we put on the right side of the table?

Second improvement

Can we get rid of the `switch`-based implementation?

- Use a look-up table

ID	Factory
GOOFY_ID	Goofy Creator
PLUTO_ID	Pluto Creator
...	...

- **Q**: What shall we put on the right side of the table?
- **A**: Creator function

Second improvement

Can we get rid of the `switch`-based implementation?

- Use a look-up table

ID	Factory
GOOFY_ID	Goofy Creator
PLUTO_ID	Pluto Creator
...	...

- **Q**: What shall we put on the right side of the table?
- **A**: Creator function

```
enum DisneyCharacterID {
        GOOFY_ID = 0,
        PLUTO_ID,
        MAX_CHARACTERS
};
//...
typedef DisneyProduct* (*creator_type_t)(void);
//...
creator_type_t lookup_table[MAX_CHARACTERS];
```

Second improvement

- **Q**: Where shall we put the look-up table?

Second improvement

- **Q**: Where shall we put the look-up table?
- **A**: In the Factory, of course.

```
class DisneyFactory {
public:
    typedef DisneyProduct* (*creator_type_t)(void);
    DisneyProduct* create(int id);
private:
    creator_type_t lookup_table[MAX_CHARACTERS];
};
```

Second improvement

- **Q**: Where shall we put the look-up table?
- **A**: In the Factory, of course.

```
class DisneyFactory {
public:
    typedef DisneyProduct* (*creator_type_t)(void);
    DisneyProduct* create(int id);
private:
    creator_type_t lookup_table[MAX_CHARACTERS];
};
```

- **Q**: Where shall we put the constructor functions?

Second improvement

- **Q**: Where shall we put the look-up table?
- **A**: In the Factory, of course.

```
class DisneyFactory {
public:
    typedef DisneyProduct* (*creator_type_t)(void);
    DisneyProduct* create(int id);
private:
    creator_type_t lookup_table[MAX_CHARACTERS];
};
```

- **Q**: Where shall we put the constructor functions?
- **A**: Either in the Factory or in the Products.

```
class Goofy : public DisneyProduct {
private:
        Goofy() : DisneyProduct("goofy") {};
        DisneyProduct* create(void)
                        { return new Goofy(); };
        friend class DisneyFactory;
};

class Pluto : public DisneyProduct {
private:
        Pluto() : DisneyProduct("pluto") {};
        DisneyProduct* create(void)
                        { return new Pluto(); };
        friend class DisneyFactory;
};
```

Second improvement

- **Q**: Where shall we put the look-up table?
- **A**: In the Factory, of course.

```
class DisneyFactory {
public:
    typedef DisneyProduct* (*creator_type_t)(void);
    DisneyProduct* create(int id);
private:
    creator_type_t lookup_table[MAX_CHARACTERS];
};
```

- **Q**: Where shall we put the constructor functions?
- **A**: Either in the Factory or in the Products.
- **Problem**: Does not compile. Why?

```
class Goofy : public DisneyProduct {
private:
        Goofy() : DisneyProduct("goofy") {};
        DisneyProduct* create(void)
                        { return new Goofy(); };
        friend class DisneyFactory;
};

class Pluto : public DisneyProduct {
private:
        Pluto() : DisneyProduct("pluto") {};
        DisneyProduct* create(void)
                        { return new Pluto(); };
        friend class DisneyFactory;
};
```

Second improvement

- **Q**: Where shall we put the look-up table?
- **A**: In the Factory, of course.

```
class DisneyFactory {
public:
    typedef DisneyProduct* (*creator_type_t)(void);
    DisneyProduct* create(int id);
private:
    creator_type_t lookup_table[MAX_CHARACTERS];
};
```

- **Q**: Where shall we put the constructor functions?
- **A**: Either in the Factory or in the Products.
- **Problem**: Does not compile. Why?
- **Solution**: The creator function must be `static`!

```
class Goofy : public DisneyProduct {
private:
        Goofy() : DisneyProduct("goofy") {};
        static DisneyProduct* create(void)
                        { return new Goofy(); };
        friend class DisneyFactory;
};

class Pluto : public DisneyProduct {
private:
        Pluto() : DisneyProduct("pluto") {};
        static DisneyProduct* create(void)
                        { return new Pluto(); };
        friend class DisneyFactory;
};
```

Second improvement

- **Q**: Where shall we put the look-up table?
- **A**: In the Factory, of course.

```
class DisneyFactory {
public:
    typedef DisneyProduct* (*creator_type_t)(void);
    DisneyProduct* create(int id);
private:
    creator_type_t lookup_table[MAX_CHARACTERS];
};
```

- **Q**: Where shall we put the constructor functions?
- **A**: Either in the Factory or in the Products.
- **Problem**: Does not compile. Why?
- **Solution**: The creator function must be `static`!

```
class Goofy : public DisneyProduct {
private:
        Goofy() : DisneyProduct("goofy") {};
        static DisneyProduct* create(void)
                        { return new Goofy(); };
        friend class DisneyFactory;
};

class Pluto : public DisneyProduct {
private:
        Pluto() : DisneyProduct("pluto") {};
        static DisneyProduct* create(void)
                        { return new Pluto(); };
        friend class DisneyFactory;
};
```

There is also another issue:

- how many instances of the Factory can we have?

Second improvement

- **Q**: Where shall we put the look-up table?
- **A**: In the Factory, of course.

```
class DisneyFactory {
public:
    typedef DisneyProduct* (*creator_type_t)(void);
    DisneyProduct* create(int id);
private:
    creator_type_t lookup_table[MAX_CHARACTERS];
};
```

- **Q**: Where shall we put the constructor functions?
- **A**: Either in the Factory or in the Products.
- **Problem**: Does not compile. Why?
- **Solution**: The creator function must be `static`!

```
class Goofy : public DisneyProduct {
private:
        Goofy() : DisneyProduct("goofy") {};
        static DisneyProduct* create(void)
                        { return new Goofy(); };
        friend class DisneyFactory;
};

class Pluto : public DisneyProduct {
private:
        Pluto() : DisneyProduct("pluto") {};
        static DisneyProduct* create(void)
                        { return new Pluto(); };
        friend class DisneyFactory;
};
```

There is also another issue:

- how many instances of the Factory can we have?
- The Factory is meant to be a singleton!

Second improvement

How to do a singleton of the `DisneyFactory` class?

FactoryExample3.h

```
class DisneyFactory {
public:
        typedef DisneyProduct* (*creator_type_t)(void);

        static DisneyProduct* create(int id);
private:
        static creator_type_t lookup_table[MAX_CHARACTERS];
};
```

FactoryExample3.cc

```
DisneyFactory::creator_type_t DisneyFactory::lookup_table[MAX_CHARACTERS] = {
        Goofy::create,  // GOOFY_ID = 0
        Pluto::create,  // PLUTO_ID = 1
};

DisneyProduct* DisneyFactory::create(int id)
{
        if (id >= MAX_CHARACTERS || id < 0)
                return NULL;
        return lookup_table[id]();
}
```

The full example is:
`./examples/09.real-application_factories-examples/FactoryExample3.cc`

Third improvement

The `enumeration`-based object identifiers are not exciting...

The full example is:
`./examples/09.real-application_factories-examples/FactoryEx`

Third improvement

The `enumeration`-based object identifiers are not exciting...

- We would like to have `string` based IDs

The full example is:
`./examples/09.real-application_factories-examples/FactoryEx`

Third improvement

The `enumeration`-based object identifiers are not exciting...

- We would like to have `string` based IDs
- Example: `"goofy"`, `"pluto"`

The full example is:
`./examples/09.real-application_factories-examples/FactoryEx`

Third improvement

The `enumeration`-based object identifiers are not exciting...

- We would like to have `string` based IDs
- Example: `"goofy"`, `"pluto"`

```
const int config[] = {GOOFY_ID, PLUTO_ID, PLUTO_ID, 1234/*invalid*/};
```

The full example is:
`./examples/09.real-application_factories-examples/FactoryEx`

Third improvement

The `enumeration`-based object identifiers are not exciting...

- We would like to have `string` based IDs
- Example: `"goofy"`, `"pluto"`

```
const char* config[] = {"goofy", "pluto", "pluto", "invalid_id"};
```

The full example is:
`./examples/09.real-application_factories-examples/FactoryEx`

Third improvement

The `enumeration`-based object identifiers are not exciting...

- We would like to have `string` based IDs
- Example: `"goofy"`, `"pluto"`

```
const char* config[] = {"goofy", "pluto", "pluto", "invalid_id"};
```

- **Q**: How do we change the look-up table?

  No silly answers: conversion function from string to enumeration.

```
static creator_type_t lookup_table[MAX_CHARACTERS];
```

The full example is:
`./examples/09.real-application_factories-examples/FactoryEx`

Third improvement

The `enumeration`-based object identifiers are not exciting...

- We would like to have `string` based IDs
- Example: `"goofy"`, `"pluto"`

```
const char* config[] = {"goofy", "pluto", "pluto", "invalid_id"};
```

- **Q**: How do we change the look-up table?

  No silly answers: conversion function from string to enumeration.

```
static creator_type_t lookup_table[MAX_CHARACTERS];
```

- **A**: Associative container: `std::map<>` of course!

```
static std::map<std::string, creator_type_t> lookup_table;
```

The full example is:
`./examples/09.real-application_factories-examples/FactoryEx`

Third improvement

FactoryExample4.cc

```
map<std::string, DisneyFactory::creator_type_t> DisneyFactory::lookup_table;

DisneyProduct* DisneyFactory::create(const string& id)
{
        if (lookup_table.empty()) {
                lookup_table["goofy"] = Goofy::create;
                lookup_table["pluto"] = Pluto::create;
        }
        if (lookup_table.find(id) == lookup_table.end())
                return NULL;
        return lookup_table[id]();
}

DisneyUser::DisneyUser()
{
        const char* config[] = {"goofy", "pluto", "pluto", "invalid_id"};

        for (size_t i = 0; i < 4; i++) {
                DisneyProduct *p = DisneyFactory::create(config[i]);
                if (p)
                        toons.push_back(p);
        }
}
```

Consideration

- What about *extensibility*?
- Adding new `DisneyProduct` requires the following modifications:

Consideration

- What about *extensibility*?
- Adding new `DisneyProduct` requires the following modifications:
  1. definition of the new `DisneyProduct` (e.g. `DonalDuck`)

Consideration

- What about *extensibility*?

- Adding new `DisneyProduct` requires the following modifications:
  1. definition of the new `DisneyProduct` (e.g. `DonalDuck`)
  2. adding a new entry in the look-up table...

```
if (lookup_table.empty()) {
    lookup_table["goofy"] = Goofy::create;
    lookup_table["pluto"] = Pluto::create;
    lookup_table["donal duck"] = DonalDuck::create;
}
```

Consideration

- What about *extensibility*?

- Adding new `DisneyProduct` requires the following modifications:
  1. definition of the new `DisneyProduct` (e.g. `DonalDuck`)
  2. adding a new entry in the look-up table...

```
if (lookup_table.empty()) {
    lookup_table["goofy"] = Goofy::create;
    lookup_table["pluto"] = Pluto::create;
    lookup_table["donal duck"] = DonalDuck::create;
}
```

  ... in the `DisneyFactory` class implementation

Consideration

- What about *extensibility*?
- Adding new `DisneyProduct` requires the following modifications:
  1. definition of the new `DisneyProduct` (e.g. `DonalDuck`)
  2. adding a new entry in the look-up table...

```
if (lookup_table.empty()) {
    lookup_table["goofy"] = Goofy::create;
    lookup_table["pluto"] = Pluto::create;
    lookup_table["donal duck"] = DonalDuck::create;
}
```

  ... in the `DisneyFactory` class implementation

- We can avoid step 2!

Exercise

Implement the Factory Method in the duck-lab.

- Start from code in `./example/DataExample5/`
- Use the factory pattern for the `Data` hirarchy

Exercise

Implement the Factory Method in the duck-lab.

- Start from code in `./example/DataExample5/`
- Use the factory pattern for the `Data` hirarchy
- Can we generalize the solution using templates?