Design Patterns

David Talby

This Lecture

- The Creational Patterns
  - Abstract Factory
  - Builder
  - Prototype
  - Factory Method
- Choosing Between Them

Creational Patterns

- Easily Change:
  - What gets created?
  - Who creates it?
  - When is it created?
- Hide the concrete classes that get created from client code
- Competing patterns, each with its own strengths

6. Abstract Factory

- A program must be able to choose one of several families of classes
- For example, a program's GUI should run on several platforms
- Each platform comes with its own set of GUI classes:

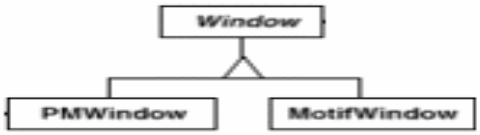
  WinButton, WinScrollBar, WinWindow

  MotifButton, MotifScrollBar, MotifWindow

  pmButton, pmScrollBar, pmWindow

The Requirements

- Uniform treatment of every button, window, etc. in the code
  - Easy - Define their interfaces:



- Uniform object **creation**
- Easy to switch between families
- Easy to add a family

The Solution

- Define a Factory - a class that creates objects:

```
class WidgetFactory {
   Button* makeButton(args) = 0;
   Window* makeWindow(args) = 0;
   // other widgets…
}
```

The Solution II

- Define a concrete factory for each of the families:

```
class WinWidgetFactory {
   Button* makeButton(args) {
     return new WinButton(args);
   }
   Window* makeWindow(args) {
     return new WinWindow(args);
   }
}
```

The Solution III

- Select once which family to use:

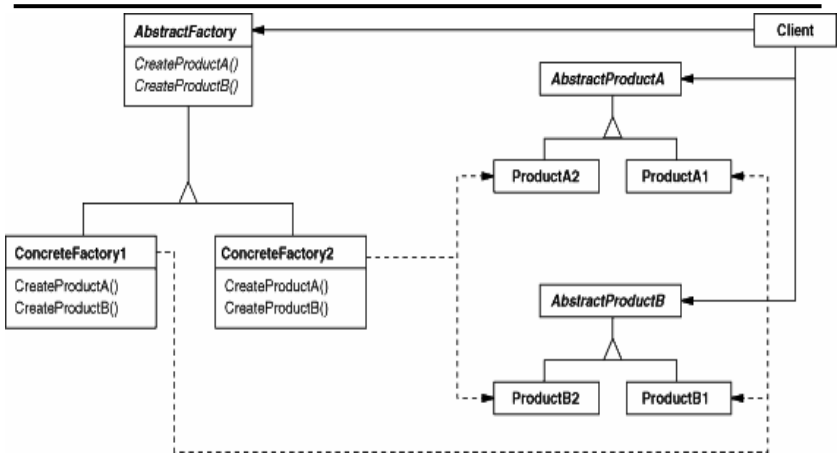
```
WidgetFactory* wf =
  new WinWidgetFactory();
```

- When creating objects in the code, don't use '*new*' but call:

```
Button* b = wf->makeButton(args);
```

- Switch families - once in the code!
- Add a family - one new factory, no effect on existing code!

The Big (UML) Picture



The Fine Print

- The factory doesn't have to be abstract, if we expect a remote possibility of having another family
- Usually one factory per application, a perfect example of a *singleton*
- Not easy to extend the abstract factory's interface

Known Uses

- Different operating systems (could be *Button*, could be *File*)
- Different look-and-feel standards
- Different communication protocols

7. Builder

- Separate the specification of how to construct a complex object from the representation of the object
- For example, a converter reads files from one file format
- It should write them to one of several output formats

The Requirements

- Single Choice Principle
  - Same reader for all output formats
  - Output format chosen once in code
- Open-Closed Principle
  - Easy to add a new output format
  - Addition does not change old code
- Dynamic choice of output format

The Solution

- We should return a different object depending on the output format:
  - *HTMLDocument*, *RTFDocument*, …
- Separate the building of the output from reading the input
- Write an interface for such a builder
- Use inheritance to write different concrete builders

The Solution II

- Here's the builder's interface:

```
class Builder {
    void writeChar(char c) { }
    void setFont(Font *f)  { }
    void newPage() { }
}
```

The Solution III

- Here's a concrete builder:

```
class HTMLBuilder
    : public Builder
{
 private:
    HTMLDocument *doc;
 public:
    HTMLDocument *getDocument() {
          return doc;
    }
    // all inherited methods here
}
```

The Solution IV

- The converter uses a builder:

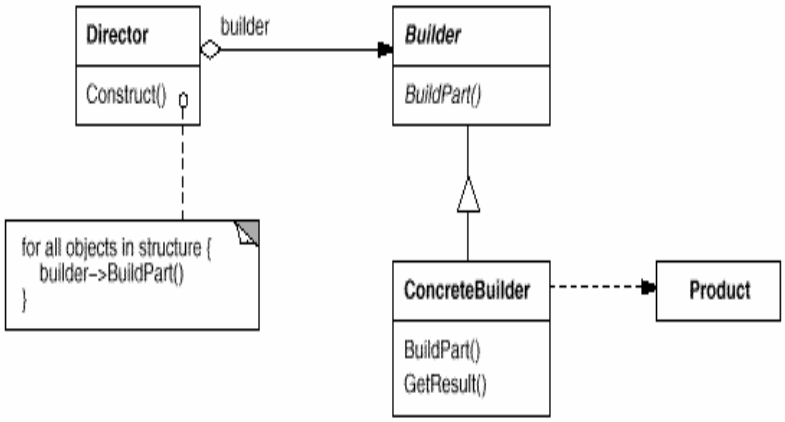
```
class Converter
{
  void convert(Builder *b) {
    while (t = read_next_token())
      switch (o.kind) {
        CHAR: b->writeChar(o);
        FONT: b->setFont(o);
        // other kinds…
      }
  }
}
```

The Solution V

- This is how the converter is used:

```
RTFBuilder *b = new RTFBuilder;
converter->convert(b);
RTFDocument *d = b->getDocument();
```

The UML



The Fine Print

- The builder's interface affects the ease of coding concrete builders
- Kinds of documents don't need a common base class
- Methods in class *Builder* are empty and not abstract
- *getResult()* is not always trivial
  - Optimizations
  - Lazy Creation

Known Uses

- Converting to different formats
- Building a parse tree in a compiler
- Building a normalized database

8. Prototype

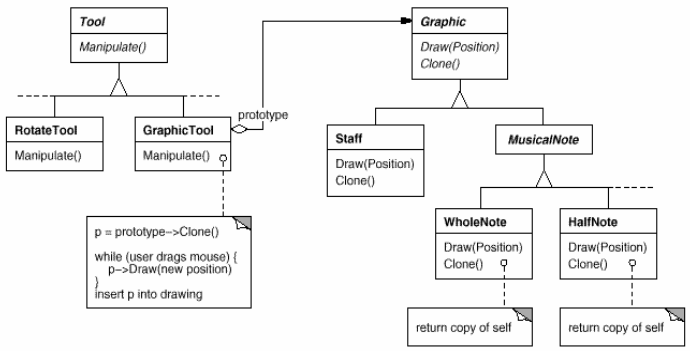
- Specify the kind of object to create using a prototypical instance
- For example, a photo/map editor has a palette of tools and objects that can be created
- How do we have only one class for creations, and parameterize it by the *class* of objects it initializes?

The Requirements

- One class for the creation tool
- Easy to add new objects
- Dynamic toolbox configuration

The Solution

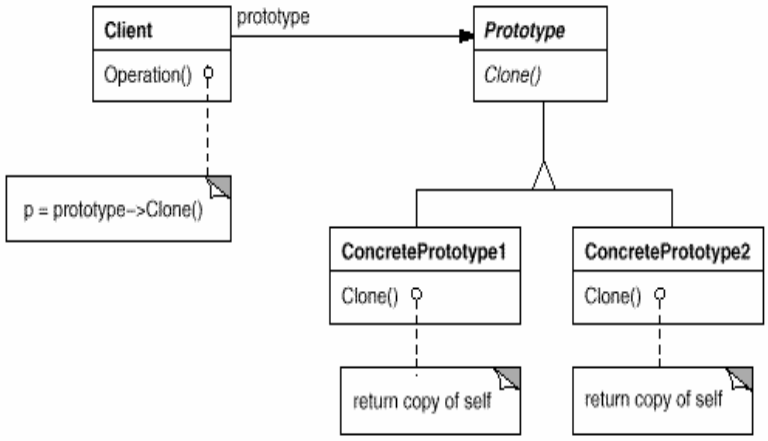
- Hold a prototype of object to create
- Creation is by cloning the prototype



The Solution II

- Less classes in the system
- Can be even less: same *Graphic* object with different properties can be used for different tools
- Tools can be chosen and configured at runtime

The UML



The Fine Print

- Prototype Manager - a runtime registry of prototype can handle dynamically linked classes
- Java, SmallTalk, Eiffel provide a default *clone()* method. C++ has copy constructors
- All of these are shallow by default
- When implementing deep clone, beware of circular references!

Known Uses

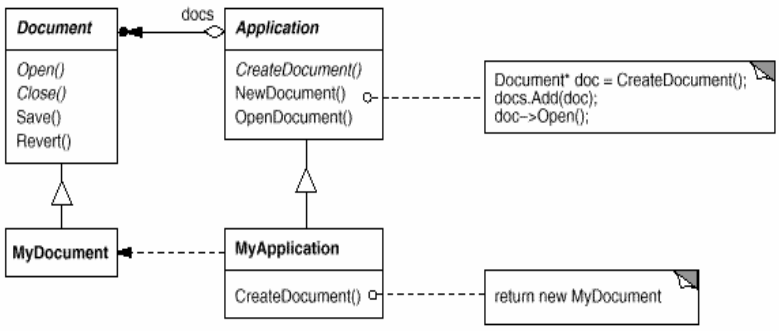
- Toolboxes / Palettes
- Supporting dynamically defined debuggers in a uniform GUI
- EJB / COM Servers
- Basically a plug-in mechanism

9. Factory Method

- Let subclasses decide which objects to instantiate
- For example, a framework for a windowing application has a class *Application* which must create an object of class *Document*
- But the actual applications and documents are not written yet!

The Solution

- Separate creation into a method



Second Variant

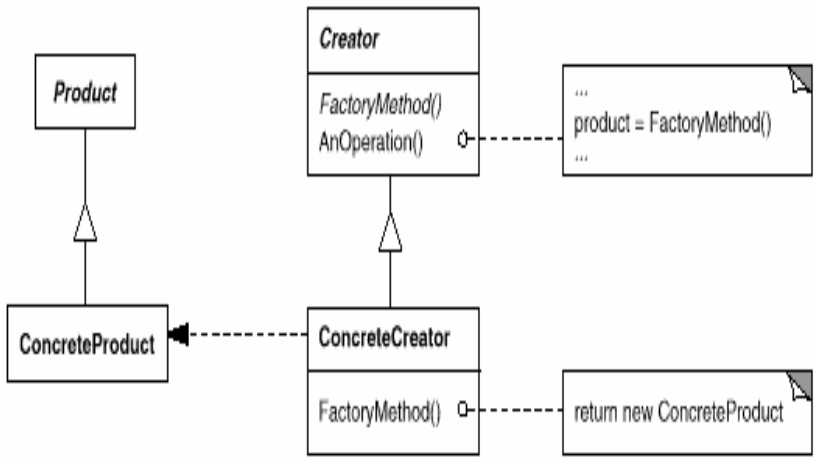
- A remote services package has a *RemoteService* class that returns objects of class *Proxy* to client
- A few clients wish to write a more potent *CachedProxy*
- How do we support this without much hassle?

Second Variant Solution

- Separate creation into a method
- *RemoteService* will have a virtual method called *CreateProxy()*
- Write *CachedProxy*, then write:

```
class CachedRemoteService
    : public RemoteService
{
  Proxy* createProxy(...) {
   return new CachedProxy(...);
  }
}
```

The UML



The Fine Print

- Two Variants: Is the factory method abstract or not?
- Good style to use factory methods even for a slight chance of need
- Parameterized factory methods make it easy to add created products without affecting old code

```
Product* createProduct(int id) {
    switch (id) { ... }
}
```

The Fine Print II

- C++ warning: You can't call a factory method from a constructor!
  - Use lazy initialization instead

```
Product* getProduct() {
  if (_product == NULL)
    _product = createProduct();
  return _product;
}
```

- Use templates to avoid subclassing
  - `Application<ExcelDocument>`
  - `complex<float>, complex<double>`

Known Uses

- A very common pattern
- Framework classes
  - *Application*, *Document*, *View*, ...
- Changing default implementations
  - *Proxy*, *Parser*, *MemoryManager*, …

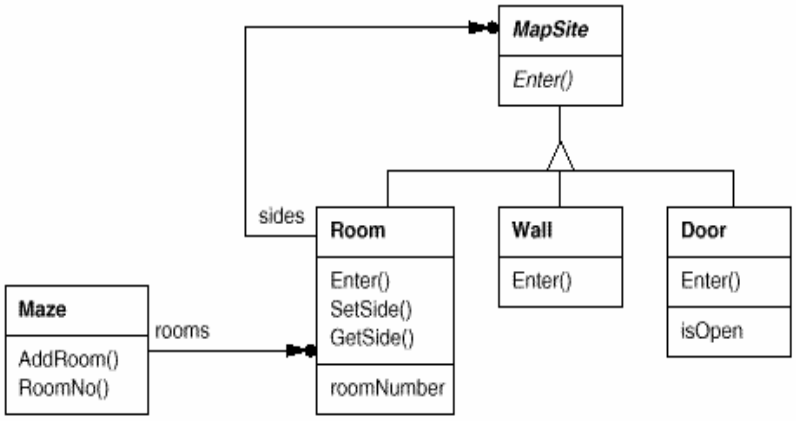
Pattern of Patterns

- Encapsulate the varying aspect
- Interfaces
- Inheritance describes variants
- Composition allows a dynamic choice between variants

Criteria for success:

Open-Closed Principle
Single Choice Principle

A Comparative Example



The Example Problem

```
Maze* MazeGame::CreateMaze () {
  Maze* aMaze = new Maze;
  Room* r1 = new Room(1);
  Room* r2 = new Room(2);
  Door* theDoor = new Door(r1, r2);

  aMaze->AddRoom(r1);
  aMaze->AddRoom(r2);

  r1->SetSide(North, new Wall);
  r1->SetSide(East, theDoor);
  // set other sides, also for r2

  return aMaze;
}
```

Enchanted Mazes

- How do we reuse the same maze with *EnchantedRoom*, *TrapDoor*?
  - Pass *createMaze* an object that can create different maze parts
  - Pass *createMaze* an object that can build a maze and then return it
  - Pass *createMaze* initialized samples of each kind of maze part
  - Move creation with *new* to other methods that descendants redefine

Abstract Factory

- Define a set of interfaces
  - *Door*, *Wall*, *Room*, ...
- Write families of classes
  - *SimpleDoor*, *SimpleRoom*, …
  - *EnchantedDoor*, *EnchantedRoom*,...
- Define an abstract *MazeFactory*, and a concrete class for each family
  - *SimpleFactory*, *EnchantedFactory*, …
- Pass *createMaze* a factory

Abstract Factory II

```
Maze* MazeGame::CreateMaze (MazeFactory*
  mf) {
  Maze* aMaze = mf->createMaze();
  Room* r1 = mf->createRoom(1);
  Room* r2 = mf->createRoom(2);
  Door* d  = mf->createDoor(r1,r2);
  // rest is same as before
```

- Families don't have to be disjoint
- Same factory can return variants of the same class

Abstract Factory Cons

- Requires a new factory class for every family
- Families are defined statically
- Parts of the complex maze are returned right after creation
- The client of the factory builds the connections between maze parts
- Maze stands for any complex object

Builder Pros & Cons

- Pros
  - Each builder can create a totally different kind of object
  - Object returned only at the end of construction - enables optimization
  - Especially if object is on network
- Cons
  - Complex Interface to builder

Prototype Pros & Cons

- Pros
  - Less Classes
  - Prototype can be customized between different creations
- Cons
  - Requires memory to hold prototype
  - Many prototypes must be passed
  - *Clone()* may be hard to implement

Factory Method P&C

- Pros
  - The simplest design
- Cons
  - Requires a new class for every change in creation
  - Compile-time choice only

The Verdict

- Use Factory Methods when there is little (but possible) chance of change
- Use Abstract Factory when different families of classes are given anyway
- Use Prototype when many small objects must be created similarly
- Use Builder when different output representations are necessary

Some Easy Cases

- Dynamic loading of classes whose objects must be created
  - only Prototype
- Creation can be highly optimized once entire structure is known
  - only Builder

Summary: Connections

- "Abstract Factories are usually implemented using Factory Methods but can also use Prototypes"
- "Builders and Abstract Factories are often Singletons"
- "Builders can use Abstract Factories to enjoy best of both worlds"