Factory Method
Factory Method

A template method for creating objects

public class Example {
    protected Bar bar() { return new Bar(); }

    public void foo() {
        blah
        Bar soap = bar();
        blah;
    }
}
Maze Game Example

- MapSite
  - Room
    - RoomWithBomb
    - EnchantedRoom
  - Wall
    - BombedWall
  - Door
    - IronDoor
    - DoorWithSpell
  - SecretPassageWall
Maze Game Example

class MazeGame{
    public Maze makeMaze() { return new Maze(); }
    public Room makeRoom(int n ) { return new Room( n ); }
    public Wall makeWall() { return new Wall(); }
    public Door makeDoor() { return new Door(); }

    public Maze CreateMaze(){
        Maze aMaze = makeMaze();
        Room r1 = makeRoom( 1 );
        Room r2 = makeRoom( 2 );
        Door theDoor = makeDoor( r1, r2);

        aMaze.addRoom( r1 );
        aMaze.addRoom( r2 );
        etc

        return aMaze;
    }
}

class BombedMazeGame extends MazeGame {
    public Room makeRoom(int n ) {
        Room r1 = makeRoom( 1 );
        Room r2 = makeRoom( 2 );
        Door theDoor = makeDoor( r1, r2);

        return new RoomWithABomb( n );
    }

    public Wall makeWall() {
        return new BombedWall();
    }
}
public class LinkedList extends Collection {
    public OrderedLinkedList() {
        this(defaultOrder());
    }

    public LinkedList(Order listOrder) {
        this(listOrder, new OrderedCollection());
    }

    public LinkedList(Collection items) {
        this(defaultOrder(), items);
    }

    protected Order defaultOrder() {
        return new RandomOrder();
    }

    public LinkedList(Order listOrder, Collection items) {
        blah
    }
}
Implementation Variation

class Hershey {

    public Candy makeChocolateStuff( CandyType id ) {
        if ( id == MarsBars ) return new MarsBars();
        if ( id == M&Ms ) return new M&Ms();
        if ( id == SpecialRich ) return new SpecialRich();

        return new PureChocolate();
    }
}

class GenericBrand extends Hershey {

    public Candy makeChocolateStuff( CandyType id ) {
        if ( id == M&Ms ) return new Flupps();
        if ( id == Milk ) return new MilkChocolate();
        return super.makeChocolateStuff(id);
    }
}
Using C++ Templates

template <class ChocolateType>
class Hershey
{
  public:
    virtual Candy* makeChocolateStuff( );
}

template <class ChocolateType>
Candy*
Hershey<ChocolateType>::makeChocolateStuff( )
{
  return new ChocolateType;
}

Hershey<SpecialRich> theBest;
Smalltalk Variant

Return the class, caller creates an object

chocolateStuff
  ^SpecialRich

some code

candy := (self chocolateStuff) new

mode code
Use Factory Method When

A class can't anticipate the class of objects it must create

A class wants its subclasses to specify the objects it creates

You want to localize the knowledge of which help classes is used in a class

But when is this?
This code requires us to send/receive data over the network, which at times is not convenient in unit tests.
Using Factory Method

public class SDWitterServer {
    public void run(int port) throws IOException {
        ServerSocket input = this.serverSocket( port );

        while (true) {
            Socket client = input.accept();
            processRequest(
                client.getInputStream(),
                client.getOutputStream());
            client.close();
        }
    }

    ServerSocket serverSocket( int port ) {
        return new ServerSocket(port);
    }

    etc.
TestServer

public class TestServer extends SDWitterServer {
    MockServerSocket testSocket;

    ServerSocket serverSocket( int port) {
        return testSocket;
    }
}

Other than using a different type of socket it performs the operations as the parent class

public class Tests extends Testcase {
    public void testLogin() {
        TestServer server = new TestServer();
        server.testSocket = new MockServerSocket("client command to login");
        server.run();
        assertTrue(server.testSocket.serverResponse() = "the correct response here");
    }
}
MockServerSocket

Returns a fake (Mock) client connection

Fakes client connection
   Does not use network
   Contains fixed requests
   Records server responses
Abstract Factory
Abstract Factory

Write a cross platform window toolkit
public void installDisneyMenu()
{
    Menu disney = new MacMenu();
disney.addItem( "Disney World" );
disney.addItem( "Donald Duck" );
disney.addItem( "Mickey Mouse" );
disney.addGrayBar( );
disney.addItem( "Minnie Mouse" );
disney.addItem( "Pluto" );
etc.
}
Use Abstract Factory

abstract class WidgetFactory {
    public Window createWindow();
    public Menu createMenu();
    public Button createButton();
}

class MacWidgetFactory extends WidgetFactory {
    public Window createWindow()
    {
        code to create a mac window
    }

    public Menu createMenu()
    {
        code to create a mac Menu
    }

    public Button createButton()
    {
        code to create a mac button
    }
}

class Win95WidgetFactory extends WidgetFactory {
    public Window createWindow()
    {
        code to create a Win95 window
    }

    public Menu createMenu()
    {
        code to create a Win95 Menu
    }

    public Button createButton()
    {
        code to create a Win95 button
    }
}
public void installDisneyMenu(WidgetFactory myFactory)
{
    Menu disney = myFactory.createMenu();
    disney.addItem( "Disney World" );
    disney.addItem( "Donald Duck" );
    disney.addItem( "Mickey Mouse" );
    disney.addGrayBar( );
    disney.addItem( "Minnie Mouse" );
    disney.addItem( "Pluto" );
    etc.
}
Abstract Factory

Encapsulate a group of individual factories that have a common theme

Separates the details of implementation of a set of objects from its general usage
How Do Abstract Factories create Things?
Use Subclass Factory Method

abstract class WidgetFactory
{
    public Window createWindow();
    public Menu createMenu();
    public Button createButton();
}

class MacWidgetFactory extends WidgetFactory
{
    public Window createWindow()
    {
        return new MacWindow();
    }

    public Menu createMenu()
    {
        return new MacMenu();
    }

    public Button createButton()
    {
        return new MacButton();
    }
}
abstract class WidgetFactory {
    private Window windowFactory;
    private Menu menuFactory;
    private Button buttonFactory;

    public Window createWindow() {
        return windowFactory.createWindow();
    }

    public Menu createMenu() {
        return menuFactory.createMenu();
    }

    public Button createButton() {
        return buttonFactory.createMenu();
    }
}

class MacWidgetFactory extends WidgetFactory {
    public MacWidgetFactory() {
        windowFactory = new MacWindow();
        menuFactory = new MacMenu();
        buttonFactory = new MacButton();
    }
}

class MacWindow extends Window {
    public Window createWindow() {
        blah
    }
}
Why Widget Factory Method?

abstract class WidgetFactory {
    private Window windowFactory;
    private Menu menuFactory;
    private Button buttonFactory;

    public Window createWindow() {
        return windowFactory.createWindow();
    }

    public Window createWindow(Rectangle size) {
        return windowFactory.createWindow(size);
    }

    public Window createWindow(Rectangle size, String title) {
        return windowFactory.createWindow(size, title);
    }

    public Window createFancyWindow() {
        return windowFactory.createFancyWindow();
    }

    public Window createPlainWindow() {
        return windowFactory.createPlainWindow();
    }
}
Use Prototype

class WidgetFactory{
    private Window windowPrototype;
    private Menu menuPrototype;
    private Button buttonPrototype;

    public WidgetFactory( Window windowPrototype,
                            Menu menuPrototype,
                            Button buttonPrototype)
    {
        this.windowPrototype = windowPrototype;
        this.menuPrototype = menuPrototype;
        this.buttonPrototype = buttonPrototype;
    }

    public Window createWindow()
    { return windowPrototype.createWindow() }

    public Window createWindow( Rectangle size)
    { return windowPrototype.createWindow( size ) }

    public Window createMenu()
    { return menuPrototype.createMenu() }

    etc.
How to prevent Cheating?

public void installDisneyMenu(WidgetFactory myFactory)
{
// We ship next week, I can't get the stupid generic Menu
// to do the fancy Mac menu stuff
// Windows version won't ship for 6 months
// Will fix this later

MacMenu disney = (MacMenu) myFactory.createMenu();
disney.addItem( "Disney World" );
disney.addItem( "Donald Duck" );
disney.addItem( "Mickey Mouse" );
disney.addMacGrayBar( );
disney.addItem( "Minnie Mouse" );
disney.addItem( "Pluto" );
etc.
}
Prototype
Prototype

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

Applicability

Use the Prototype pattern when

A system should be independent of how its products are created, composed, and represented; and

When the classes to instantiate are specified at run-time; or

To avoid building a class hierarchy of factories that parallels the class hierarchy of products; or

When instances of a class can have one of only a few different combinations of state.
Insurance Example

Insurance agents start with a standard policy and customize it

Two basic strategies:

Copy the original and edit the copy

Store only the differences between original and the customize version in a decorator
Copying Issues

Shallow Copy Verse Deep Copy

Original Objects

Shallow Copy
Shallow Copy Verse Deep Copy

Original Objects

Deep Copy

Deeper Copy
Cloning Issues - C++ Copy Constructors

class Door {
    public:
        Door();
        Door( const Door&);
        virtual Door* clone() const;

        virtual void Initialize( Room*, Room* );
        // stuff not shown
    private:
        Room* room1;
        Room* room2;
    }

Door::Door ( const Door& other ) // Copy constructor {
    room1 = other.room1;
    room2 = other.room2;
}

Door* Door::clone() const {
    return new Door( *this );
}
Cloning Issues - Java Clone

**Shallow Copy**

class Door implements Cloneable {
    private Room room1;
    private Room room2;

    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

**Deep Copy**

class Door implements Cloneable {
    private Room room1;
    private Room room2;

    public Object clone() throws CloneNotSupportedException {
        Door thisCloned = (Door) super.clone();
        thisCloned.room1 = (Room) room1.clone();
        thisCloned.room2 = (Room) room2.clone();
        return thisCloned;
    }
}
Prototype-based Languages

No classes

Behaviour reuse (inheritance)
  Cloning existing objects which serve as prototypes

Some Prototype-based languages

  Self
  JavaScript
  Squeak (eToys)
  Perl with Class::Prototyped module