12 Challenge 1 — Multi-Pattern Design Challenge

**Scenario** Design a messaging platform that can:

* create messages in several formats (SMS, Email, Push) with the option to add more later;
* minimise allocation overhead by re-using message objects rather than instantiating afresh every time.

You must combine **two creational patterns** to satisfy both goals.

#### **Tasks**

1. analysis/design\_choice.md Explain **which** two patterns you will combine and **why** each one is needed for a) extensibility of message types and b) runtime reuse of heavy objects.
2. Model the domain under src/main/java/clean/  
   * **Abstract Factory** layer  
     + Message (product interface) and concrete products SmsMessage, EmailMessage, PushMessage.
     + MessageFactory (abstract creator) plus concrete factories SmsFactory, EmailFactory, PushFactory.
   * **Object Pool** layer  
     + Generic ObjectPool<T> and a concrete MessagePool that pools Message instances per type.
3. Implement MessagingService that, on send(type, payload),  
   * acquires a message from the correct pool (creates one via its factory if pool is empty);
   * injects the payload, “sends” (prints to console), then returns the instance to the pool.
4. Unit tests under src/test/java/clean/  
   * FactoryProducesRightProductTest – ensure each factory returns the expected subtype.
   * PoolReusesInstancesTest – send the same message type twice, assert both calls received the **same** object identity.
   * ThreadSafetyTest – spin 20 threads against the pool of size 5; assert no null is returned.
5. reflection.md  
   * Trade-offs of mixing patterns (extra complexity, pool eviction policy).
   * How you would add a new format (e.g. WhatsApp) with zero changes to existing classes.

#### **Deliverables**

analysis/design\_choice.md

src/main/java/clean/factory/MessageFactory.java

src/main/java/clean/factory/SmsFactory.java

src/main/java/clean/factory/EmailFactory.java

src/main/java/clean/factory/PushFactory.java

src/main/java/clean/message/Message.java

src/main/java/clean/message/SmsMessage.java

src/main/java/clean/message/EmailMessage.java

src/main/java/clean/message/PushMessage.java

src/main/java/clean/pool/ObjectPool.java

src/main/java/clean/pool/MessagePool.java

src/main/java/clean/service/MessagingService.java

src/test/java/clean/\*\* ← three tests

reflection.md

README.md ← build & run instructions

## **Detailed Solution**

### **1 Design Choice (Abstract Factory + Object Pool)**

* **Abstract Factory** supplies a family of related products (messages) through one façade. Adding WhatsApp later means creating WhatsAppMessage + WhatsAppFactory; the client never changes.
* **Object Pool** keeps a bounded cache of message instances keyed by type. High-volume SMS blasts recycle SmsMessage objects, cutting GC churn and allocation cost.

### **2 Core Code**

#### **Message hierarchy**

/\* message/Message.java \*/

package clean.message;

public interface Message {

void setPayload(String body);

void send();

}

/\* message/SmsMessage.java \*/

package clean.message;

public class SmsMessage implements Message {

private String body;

public void setPayload(String body){ this.body = body; }

public void send(){ System.out.println("[SMS] " + body); }

}

/\* message/EmailMessage.java \*/

package clean.message;

public class EmailMessage implements Message {

private String body;

public void setPayload(String body){ this.body = body; }

public void send(){ System.out.println("[EMAIL] " + body); }

}

/\* message/PushMessage.java \*/

package clean.message;

public class PushMessage implements Message {

private String body;

public void setPayload(String body){ this.body = body; }

public void send(){ System.out.println("[PUSH] " + body); }

}

#### **Abstract Factory layer**

/\* factory/MessageFactory.java \*/

package clean.factory;

import clean.message.Message;

public interface MessageFactory { Message create(); }

/\* factory/SmsFactory.java \*/

package clean.factory;

import clean.message.\*;

public class SmsFactory implements MessageFactory {

public Message create(){ return new SmsMessage(); }

}

/\* factory/EmailFactory.java \*/

package clean.factory;

import clean.message.\*;

public class EmailFactory implements MessageFactory {

public Message create(){ return new EmailMessage(); }

}

/\* factory/PushFactory.java \*/

package clean.factory;

import clean.message.\*;

public class PushFactory implements MessageFactory {

public Message create(){ return new PushMessage(); }

}

#### **Generic Pool**

/\* pool/ObjectPool.java \*/

package clean.pool;

import java.util.\*;

public abstract class ObjectPool<T> {

private final Queue<T> idle = new ArrayDeque<>();

private final int max;

protected ObjectPool(int max){ this.max = max; }

protected abstract T newObject();

public synchronized T acquire(){

if(!idle.isEmpty()) return idle.poll();

return idle.size()+busyCount() < max ? newObject() : null;

}

public synchronized void release(T obj){ idle.offer(obj); }

protected int busyCount(){ return max - idle.size(); }

}

#### **Message-specific Pool**

/\* pool/MessagePool.java \*/

package clean.pool;

import clean.factory.\*;

import clean.message.Message;

import java.util.Map;

import java.util.concurrent.ConcurrentHashMap;

/\*\* One pool per message type, keyed by factory class. \*/

public final class MessagePool {

private final int size;

private final Map<Class<? extends MessageFactory>, ObjectPool<Message>> pools =

new ConcurrentHashMap<>();

public MessagePool(int size){ this.size = size; }

public Message acquire(MessageFactory f){

ObjectPool<Message> p = pools.computeIfAbsent(

f.getClass(),

k -> new ObjectPool<Message>(size){ protected Message newObject(){ return f.create(); }}

);

return p.acquire();

}

public void release(MessageFactory f, Message m){

pools.get(f.getClass()).release(m);

}

}

#### **Messaging Service (client façade)**

/\* service/MessagingService.java \*/

package clean.service;

import clean.factory.\*;

import clean.message.\*;

import clean.pool.MessagePool;

public class MessagingService {

private final MessagePool pool = new MessagePool(5); // max 5 per type

public void send(MessageFactory factory, String payload){

Message msg = pool.acquire(factory);

if(msg == null){

System.out.println("Pool exhausted for "+factory.getClass().getSimpleName());

return;

}

try{

msg.setPayload(payload);

msg.send();

}finally{

pool.release(factory, msg);

}

}

}

#### **Main demo**

package clean;

import clean.factory.\*;

import clean.service.MessagingService;

public class Main {

public static void main(String[] args){

MessagingService svc = new MessagingService();

svc.send(new SmsFactory(), "Hi by SMS");

svc.send(new EmailFactory(), "Monthly report attached");

svc.send(new PushFactory(), "You have a new follower");

}

}

### **3 Test Snippets**

/\* FactoryProducesRightProductTest \*/

assertTrue(new SmsFactory().create() instanceof SmsMessage);

/\* PoolReusesInstancesTest \*/

MessagePool p = new MessagePool(1);

MessageFactory f = new SmsFactory();

Message a = p.acquire(f);

p.release(f,a);

Message b = p.acquire(f);

assertSame(a,b); // reused instance

/\* ThreadSafetyTest \*/

Executors.newFixedThreadPool(10).invokeAll(

IntStream.range(0,10).mapToObj(i ->

(Callable<Boolean>) () -> {

Message m = p.acquire(f);

Thread.sleep(10);

p.release(f,m);

return m != null;

}).collect(Collectors.toList()));

### **Reflection (high-level summary)**

| **Aspect** | **Prototype + Object Pool** | **Abstract Factory + Object Pool (your design)** |
| --- | --- | --- |
| **Extensibility** | Cloning avoids constructor logic but still needs template setup. | Plug-in a new factory & product; pools remain unchanged. |
| **Resource reuse** | Prototype clones still allocate memory. | Pool reuses exact instances, cutting GC time. |
| **Risk** | Hidden shared state in cloned objects. | Potential starvation if pool size too small; needs metrics. |

By composing **Abstract Factory** for variability **and** **Object Pool** for efficiency, the messaging system stays open for new formats yet avoids needless object churn under heavy load.