IBIMA Publishing
Communications of International Proceedings
https://ibimapublishing.com/p-articles/45SE/2025/4515525/
Vol. 2025 (22), Article ID 4515525

Workflow Patterns in Business Process Modelling and Implementation*

Marcin SIRANT and Jaroslaw KOSZELA

Military University of Technology, Warsaw, Poland

Correspondence should be addressed to: Marcin SIRANT, marcin.sirant@student.wat.edu.pl

* Presented at the 45th IBIMA International Conference, 25-26 June 2025, Cordoba, Spain

Abstract

Business process modelling enables organizations to understand and optimize workflows by creating visual, data-driven representations of key business operations. However, accurately reflecting real-world business processes requires sophisticated mechanisms and the careful selection of workflow patterns. This article analyzes fundamental workflow patterns, discusses their implementation in various programming languages. A comparative analysis highlights the practical challenges and trade-offs encountered when implementing these patterns. The work aims to guide engineers in developing efficient business modelling tools and optimizing workflow execution.

Keywords: Business process modelling, Workflow patterns, Process optimization

Introduction

Over the years, the workflow management approach and technology have been applied to many corporate information systems (Jablonski and Bussler, 1996; Leymann and Roller, 2000; van der Aalst, Desel and Oberweis, 2000; Marinescu, 2002; van der Aalst and van Hee, 2002). Workflow management systems like Staffware, IBM MQSeries Workflow, Camunda, Aurea BPM provide versatile tools for modelling and executing structured business processes (Deborin et al., 2002; Berti et al., 2020; Waszkowski, 2022). Some of them have more patterns to use, while others are more modest, providing only fundamental templates (David et al., 2023). In the realm of business process modelling, it is crucial to delineate the fundamental requirements that consistently arise. Some researchers emphasize that models should remain as simple as possible to ensure maximum flexibility (Berti et al., 2020). Others advocate for sophisticated methods to facilitate workflow evolution and the transition of cases between different workflow models (Deborin et al., 2002; van der Aalst, Weske and Grünbauer, 2005). Research provides valuable insights into these requirements and their implications for effective business process modelling (Russell and Hofstede, 2006).

The article focuses on presenting workflow patterns in business process modelling and demonstrates how these patterns can be implemented in various programming languages. The aim is to highlight potential problems and provide examples of usage. It is important to remember that every command is ultimately executed by the computer. While we operate at a high level of abstraction, each action is translated into computer language. Each block has its own implementation, which can be complex and may vary across programming languages. What is simple in one language can be challenging in another, as every language has its unique set of issues.

Cite this Article as: Marcin SIRANT and Jaroslaw KOSZELA, Vol. 2025 (22) "Workflow Patterns in Business Process Modelling and Implementation " Communications of International Proceedings, Vol. 2025 (22), Article ID 4515525, https://doi.org/10.5171/2025.4515525

Workflow Patterns

Sequence Pattern

This is a common pattern that appears in most programming languages. Data is processed sequentially, meaning that after one task is completed, the data is processed in the next task. Sequence is fundamental building block and is used to create a series of tasks that perform one after another.

Figure 1 illustrates the sequence pattern where blocks are executed in order from 1 to 3.

Figure 1 Sequence pattern

Example:

- Transferring money: The system deducts the amount, then confirms the transaction.

Implementation:

Java	Python

public class SequencePattern {
  public void transfer(double amount) {
  deduct(amount);
  confirm(amount);
  }
  private void deduct(double amount) {
  // subtract amount from account
  }
  private void confirm(double amount) {
  // send confirmation notification
  }
}
	class SequencePattern:
	  def transfer(self, amount):
	  self.deduct(amount)
	  self.confirm(amount)
	
	  def deduct(self, amount):
	  # subtract amount from account
	  pass
	
	  def confirm(self, amount):
	  # send confirmation notification
	  pass

Listing 1 Example of sequence pattern in java and python

The code models a straightforward linear workflow: it deducts an amount from an account and then sends a confirmation. Each step is encapsulated in its own method (Java) or function (Python), ensuring clear task separation. Java's static typing and method declarations make the sequence explicit and self-documenting, while Python's dynamic typing yields more concise code. Performance for simple, single-threaded sequences is comparable, though Java may edge ahead in compute-intensive scenarios due to JVM optimizations.

Problems:

When two threads run a task at the same time, race conditions can occur, which can manipulate the data we are processing, which is unwanted. The propose of some solutions to take a pre-emptive approach by implementing a safe process model that does not allow another task to run until the last task is finished.

Parallel Split Pattern

Branch divergence into two or more parallel branches, each of them is executed simultaneously. This pattern provides data processing at the same time.

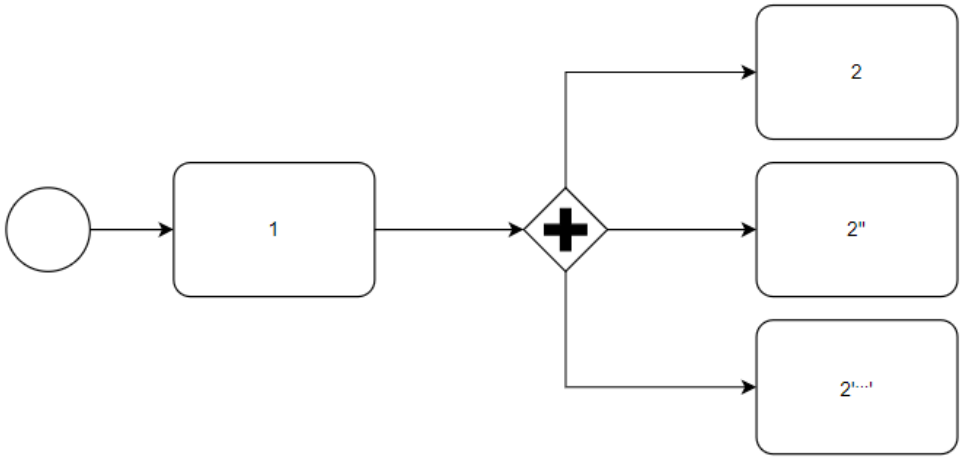

Figure 2 Parallel Split pattern

Example:

- During the construction of a new residential development, instead of waiting for each house to be built one by one, the developer decides to work on several houses simultaneously.

Implementation:

Java	Python

```java
public class ParallelSplit {
   public static void main(String[] args) throws
InterruptedException {
      ExecutorService exec =
Executors.newFixedThreadPool(2);
      exec.submit(() -> build("House A"));
      exec.submit(() -> build("House B"));
      exec.shutdown();
      exec.awaitTermination(1,
TimeUnit.MINUTES);
   }
   private static void build(String name) { /*
construction logic */ }
}
```

```python
def build(name): pass

with ThreadPoolExecutor(max_workers=2) as
exec:
   exec.submit(build, "House A")
   exec.submit(build, "House B")
```

Listing 2 Example of parallel split pattern in java and python

Tasks are dispatched concurrently using thread pools. Both implementations wait for completion before exiting. Java's ExecutorService achieves true parallelism across cores; Python's threads contend under the GIL, making them ideal for I/O-bound work. Java code is more verbose but delivers consistent multi-core performance.

Problems:

When multiple branches execute concurrently, there may be a need for synchronization mechanisms to ensure that shared resources are accessed safely and consistently.

Synchronization Pattern

Synchronization is a process in which several branches merge to form a single branch. Only when all input branches are completed will the next task be performed.

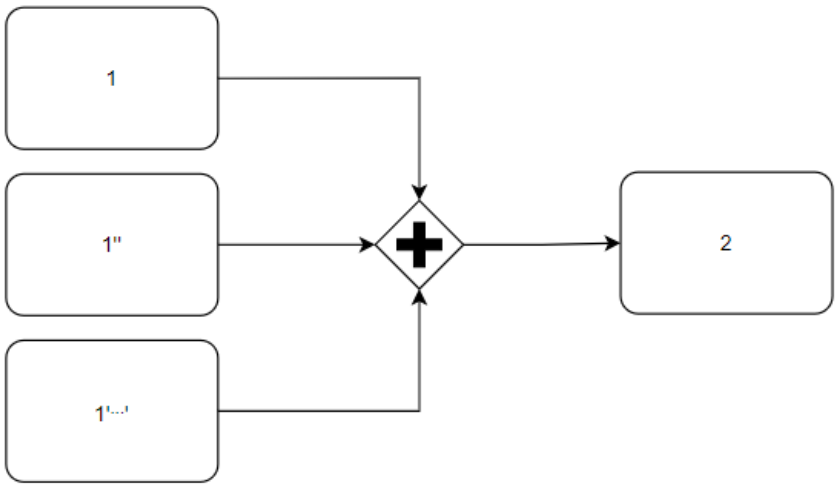

Figure 3 Synchronization pattern

Example:

- Preparing fixed meal courses; serve only once all are ready.

Implementation:

Java	Python

public class Synchronization {
  public static void main(String[] args) throws InterruptedException {
  String[] parts = {"Appetizer","Main","Dessert"};
  CountDownLatch latch = new CountDownLatch(parts.length);
  ExecutorService exec = Executors.newFixedThreadPool(parts.length);
  for (String p: parts) exec.submit(() -> { prep(p); latch.countDown(); });
  latch.await();
  serve();
  exec.shutdown();
  }
  private static void prep(String p) { /* prep logic */ }
  private static void serve() { /* serve logic */ }
}
	parts = ["Appetizer","Main","Dessert"]
	barrier = threading.Barrier(len(parts)+1)
	
	def prep(p):
	  pass
	  barrier.wait()
	
	for p in parts:
	  threading.Thread(target=prep, args=(p,)).start()
	
	barrier.wait()
	def serve(): pass
	serve()

Listing 3 Example of synchronization pattern in java and python

Uses a latch/barrier to block until all preparatory tasks signal completion, then proceeds. Java's CountDownLatch offers flexible countdown semantics; Python's Barrier requires an extra slot for the main thread. Java scales better under high contention.

Problems

If one of the incoming branches is unable to be completed for a merge design, the use of the synchronization pattern can cause a deadlock. This can occur as a result of a design error, failure to complete one of the tasks on the branch (perhaps due to an exception).

Exclusive Choice Pattern

Exclusive Choice occurs when a single branch splits into two or more separate branches. When the initial branch activates, control is instantly transferred to only one of the new branches. This transfer is governed by a mechanism that chooses which specific branch to follow among the available options.

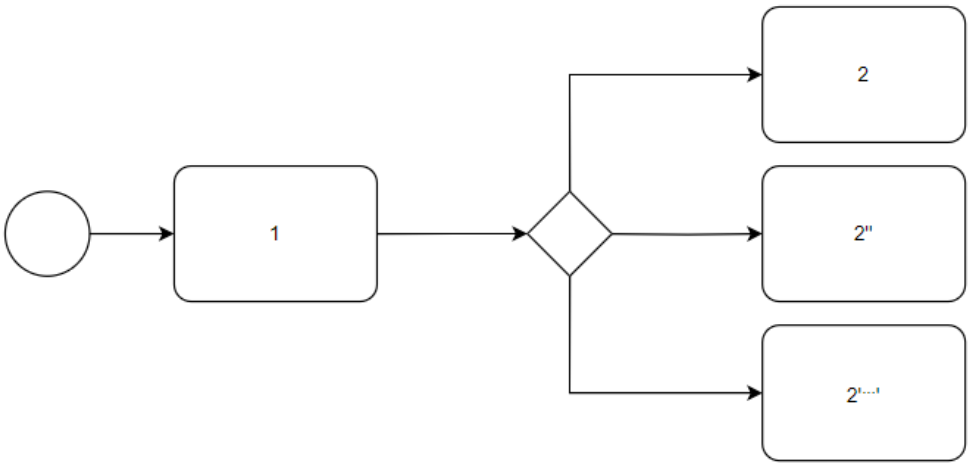

Figure 4 Exclusive Choice pattern

Example:

- Depending on the amount of the apartment purchase, customers can receive various discounts: 5% for purchases between 100,000 and 200,000, 10% for purchases between 200,000 and 500,000, and 15% for purchases over 500,000.

Implementation:

Java	Python

public class ExclusiveChoice {
  public double discount(double amt) {
  if (amt > 500 000) return 0.15;
  else if (amt > 200_000) return 0.10;
  else if (amt >= 100 000) return 0.05;
  else return 0.0;
  }
}
	def discount(amt):
	  if amt > 500 000: return 0.15
	  elif amt > 200 000: return 0.10
	  elif amt >= 100_000: return 0.05
	  else: return 0.0

Listing 4 Example of exclusive choice pattern in java and python

Chooses exactly one branch per invocation based on conditions. Both use familiar conditional constructs. Java's compile-time checks prevent missing cases; Python's runtime checks are more flexible. Performance differences are negligible.
Problems:

The main challenge in this problem is to ensure that all possible cases are handled correctly and to safety against undesirable situations, such as errors in decision logic, which can lead to unpredictable results.

Simple Merge Pattern

Combining two or more branches into a single branch such that the thread of control is passed to the next branch with each completion of the incoming branch.

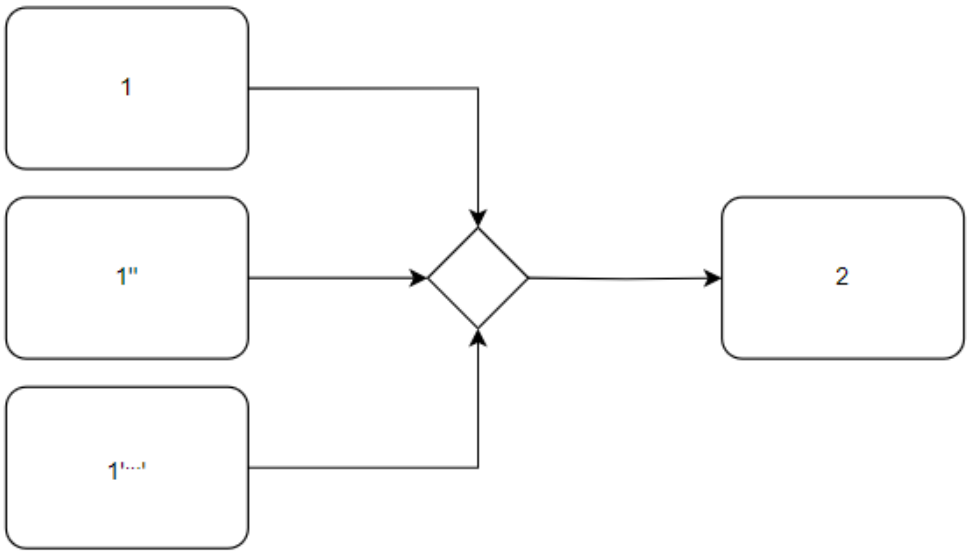

Figure 5 Simple Merge pattern

Example:

- Perform cleaning of the tool after using the pan or pot.

Implementation:

Java	Python

public class SimpleMerge {
  public static void main(String[] args) throws Exception {
  ExecutorService exec = Executors.newCachedThreadPool();
  CompletionService<Void> cs = new ExecutorCompletionService<>(exec);
  cs.submit(() -> { use("pan"); return null; });
  cs.submit(() -> { use("pot"); return null; });
  cs.take();
  clean();
  exec.shutdown();
  }
  private static void use(String t) { /* use logic */ }
  private static void clean() { /* clean logic */ }
}
	def use(t): pass
	def clean(): pass
	
	with ThreadPoolExecutor() as exec:
	  futures = [exec.submit(use, t) for t in ("pan","pot")]
	  done, = wait(futures, return when=FIRST COMPLETED)
	clean()

Listing 5 Example of simple merge in java and python

Merges on the first completing branch, triggering the next task immediately. Java's CompletionService cleanly captures the first result; Python's wait(..., FIRST_COMPLETED) is slightly more verbose. Both yield similar latency characteristics.

Problems:

When the safety of the entering location to the merging (XOR) cannot be guaranteed, there may be problems when using this pattern.

Multi-Choice Pattern

A branch splits into two or more branches. As a result, upon activation of the incoming branch, control is instantly transferred to one or more of the outgoing branches, chosen by a mechanism.

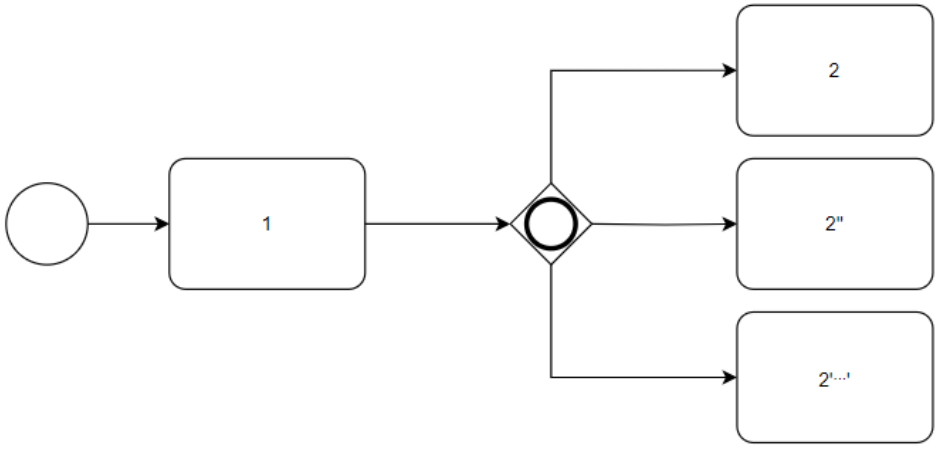

Figure 6 Multi-Choice pattern

Example:

- In the event of a fire at a construction site, the system automatically notifies the fire department and emergency services.

Implementation:

Java	Python

public void onFire() {
  Thread fireThread = new Thread(this::notifyFireDept);
  Thread emsThread = new Thread(this::notifyEMS);
 	
  fireThread.start();
  emsThread.start();
}
 	
  private void notifyFireDept() {
  // logic for notifying fire department
  System.out.println("Fire department notified");
  }
 	
  private void notifyEMS() {
	class MultiChoice:
	  def on fire(self):
	  # Create threads for each notification
	  fire_thread = threading.Thread(target=self.notify_fire_dept)
	  ems_thread = threading.Thread(target=self.notify_ems)
	
	  # Start threads
	  fire_thread.start()
	  ems_thread.start()
	
	  def notify_fire_dept(self):
	  print("Fire department notified")
	
	  def notify_ems(self):
	  print("EMS notified")

```
        // logic for notifying EMS
        System.out.println("EMS notified");
    }

    public static void main(String[] args) {
        new MultiChoice().onFire();
    }
}
```

```
# Example usage
if __name__ == "__main__":
    multi_choice = MultiChoice()
    multi_choice.on_fire()
```

Listing 6 Example of multi-choice pattern in java and python

Activates multiple outgoing branches simultaneously. Structural similarity in both languages; Java enforces encapsulation via classes, Python offers terseness. Threads are used to parallelize notifications.

Problems:

This design has been shown to have two problems. First, the same problem that happens when using the Exclusive Choice also occurs when using this pattern: making sure that at least one outgoing branch is chosen from the available alternatives. There is a chance that the procedure will stall if this is not the case. Second, the question of whether there are any indirect ways to achieve the same behaviour arises when an offering does not explicitly support the Multi-Choice architecture.

Structured Synchronizing Merge Pattern

When two or more branches (which separated earlier at a unique point) come back together into one later branch, control is handed over to the new branch once all active branches have been enabled. This is called a Structured Synchronizing Merge. This merge happens in a structured context, meaning there has to be a single Multi-Choice construct earlier in the process model that the Structured Synchronizing Merge is connected to. It must merge all the branches coming from the Multi-Choice. These branches should either flow from the Structured Synchronizing Merge without any splits or joins, or they should be structured in form (meaning, balanced splits and joins).

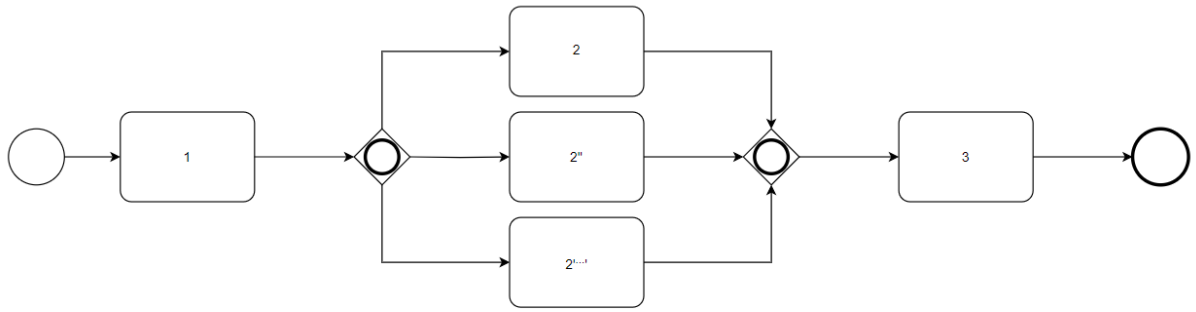

Figure 7 Structured Synchronizing Merge pattern

Examples:

- E-commerce shopping process. Customer selects multiple items (diverging branches), next they proceed to a single checkout process (structured synchronizing merge). The checkout will only be enabled when all selected items (active incoming branches) are confirmed.

Implementation:

Java	Python
public class StructSyncMerge {	items = ["A","B","C"]

```
   public static void main(String[] args) throws
InterruptedException {
      String[] items = {"A","B","C"};
      CountDownLatch latch = new
CountDownLatch(items.length);
      ExecutorService exec =
Executors.newFixedThreadPool(items.length);
      for (String i: items) exec.submit(() -> {
confirm(i); latch.countDown(); });
      latch.await();
      checkout();
      exec.shutdown();
   }
   private static void confirm(String i) { /* logic */
}
   private static void checkout() { /* logic */ }
}
```

```
barrier = threading.Barrier(len(items)+1)

def confirm(i):
   pass
   barrier.wait()

for i in items:
   threading.Thread(target=confirm,
args=(i,)).start()

barrier.wait()
def checkout(): pass
checkout()
```

Listing 7 Example of structured synchronizing merge pattern in java and python

Merges only after all active branches (originating from a prior multi-choice) complete. Java's latch integrates seamlessly with executors; Python's barrier can deadlock if miscounted.

Problems:

Structured Synchronizing Merge is not suitable for use in contexts that lack a structured form.

Multi-Merge Pattern

The merging of two or more branches into a single branch, where each activation of an incoming branch passes control to the subsequent branch.

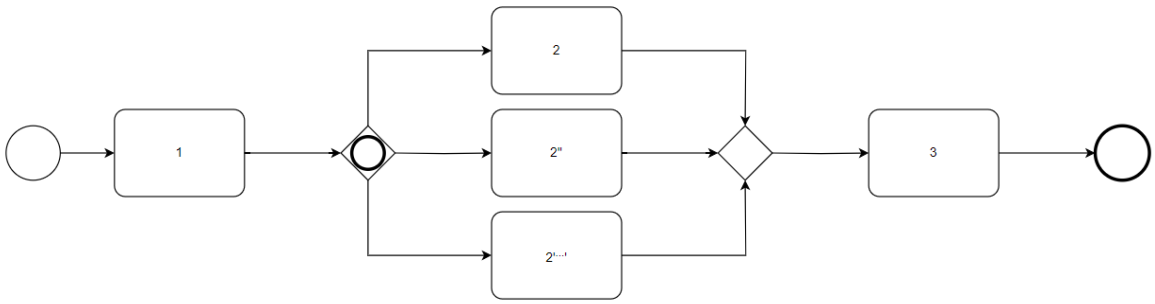

Figure 8 Multi-Merge pattern

Example:

- In software development, multiple developers often work on different features (branches) of the same project concurrently. Once the work on a feature (branch) is complete, it gets merged (converges) back

into the main codebase (subsequent branch). This process repeats for each feature (branch), with the main codebase being the subsequent branch that receives the thread of control after each merge.

Implementation:

Java	Python

public class MultiMerge {
  public static void main(String[] args) throws InterruptedException {
  String[] feats = {"Login","Search","Checkout"};
  ExecutorService exec = Executors.newFixedThreadPool(feats.length);
  for (String f: feats) exec.submit(() -> { develop(f); merge(f); });
  exec.shutdown();
  exec.awaitTermination(1, TimeUnit.MINUTES);
  }
  private static void develop(String f) { /* dev logic */ }
  private static void merge(String f) { /* merge logic */ }
}
	feats = ["Login","Search","Checkout"]
	
	def dev_and_merge(f):
	  develop(f)
	  merge(f)
	
	def develop(f): pass
	def merge(f): pass
	
	threads = [threading.Thread(target=dev_and_merge, args=(f,)) for f in feats]
	for t in threads: t.start()
	for t in threads: t.join()

Listing 8 Example of multi merge pattern in java and python

Each branch independently merges back into the main thread of control upon completion. Java's executor framework offers controlled shutdown and error handling; Python's raw threads yield more concise code but require manual lifecycle management. Executors simplify resource management for complex merge scenarios.

Problems:

Managing dependencies between the merged branches can become complex, especially if there are interdependencies or conflicting changes between them.

Conclusion

This article provides an overview of the basic workflow patterns with implementations in Java, Python, and discusses where you may encounter problems when using them. Understanding workflow patterns in business processes modelling is crucial for building solid and efficient systems. Thanks to use them, systems can be more integrated and intuitive for people who use products and future programmers who will be required to develop the software.

It is worth noting that these patterns are not related only to specific programming languages.
The examples show implementations in two programming languages in which these patterns are presented. Each programming language has its own nuances and is designed/used for different things. Thus, each pattern may be faster or slower in some languages.

While these patterns provide a solid foundation, it is necessary to adapt and improvise according to the unique requirements and constraints of each business process. The goal should always be to make the process more efficient and effective, and sometimes this may require creative interpretation and application of these patterns.

The work aims to serve as a foundation for further exploration of patterns in distributed systems and to identify novel concepts for optimizing database queries in distributed database environments.

References

- van der Aalst, W., Desel, J. and Oberweis, A. (eds) (2000) Business Process Management. Berlin, Heidelberg: Springer Berlin Heidelberg. Available at: https://doi.org/10.1007/3-540-45594-9.
- van der Aalst, W.M.P. and van Hee, K. (2002) Workflow Management. The MIT Press. Available at: https://doi.org/10.7551/mitpress/7301.001.0001.
- van der Aalst, W.M.P., Weske, M. and Grünbauer, D. (2005) 'Case handling: a new paradigm for business process support', Data & Knowledge Engineering, 53(2), pp. 129–162. Available at: https://doi.org/https://doi.org/10.1016/j.datak.2004.07.003.
- Berti, A. et al. (2020) 'An Open-Source Integration of Process Mining Features Into the Camunda Workflow Engine: Data Extraction and Challenges', ArXiv, abs/2009.06209. Available at: https://api.semanticscholar.org/CorpusID:221655650.
- David, I. et al. (2023) 'Blended modeling in commercial and open-source model-driven software engineering tools: A systematic study', Software and Systems Modeling, 22(1), pp. 415–447. Available at: https://doi.org/10.1007/s10270-022-01010-3.
- Deborin, E. et al. (2002) 'Continuous business process management with holosofx bpm suite and ibm mqseries workflow', in. Available at: https://api.semanticscholar.org/CorpusID:59734101.
- Jablonski, S. and Bussler, C. (1996) Workflow management - modeling concepts, architecture and implementation. International Thomson.
- Leymann, F. and Roller, D. (2000) Production workflow - concepts and techniques.
- Marinescu, D. (2002) Internet-Based Workflow Management—Toward a Semantic Web.
- Russell, N. and Hofstede, A.H.M. (2006) 'WORKFLOW CONTROL-FLOW PATTERNS A Revised View', Business, 2. Available at: https://doi.org/10.1.1.93.6974.
- Waszkowski, R. (2022) 'Low-code Development Platform for Business Process Automation: Aurea BPM', in. Available at: https://doi.org/10.54941/ahfe1001633.