

Modeling software applications with Spring Statemachine (SSM)

Host: Popovici Gabriel
Division: Domains
Hub: Bucharest,Romania

Workshop Agenda

- ▶ **Prerequisites**
 - ▶ **Using SSM 3.0.0.M1**
 - ▶ **Java with JDK 8**
 - ▶ **Spring Framework 5.2.0.RC1**
 - ▶ **Zookeeper**
 - ▶ **Toolchain: Maven 3.5.x**
 - ▶ **Clone <https://github.com/popovici-gabriel/contact-service>**
- ▶ **Context**
 - ▶ **Types of systems modeling**
 - ▶ **Workflow engine vs Statemachine**
 - ▶ **Building blocks of Statemachine**
- ▶ **SSM Concepts**
 - ▶ **What?**
 - ▶ **Why?**
 - ▶ **Hands-on**

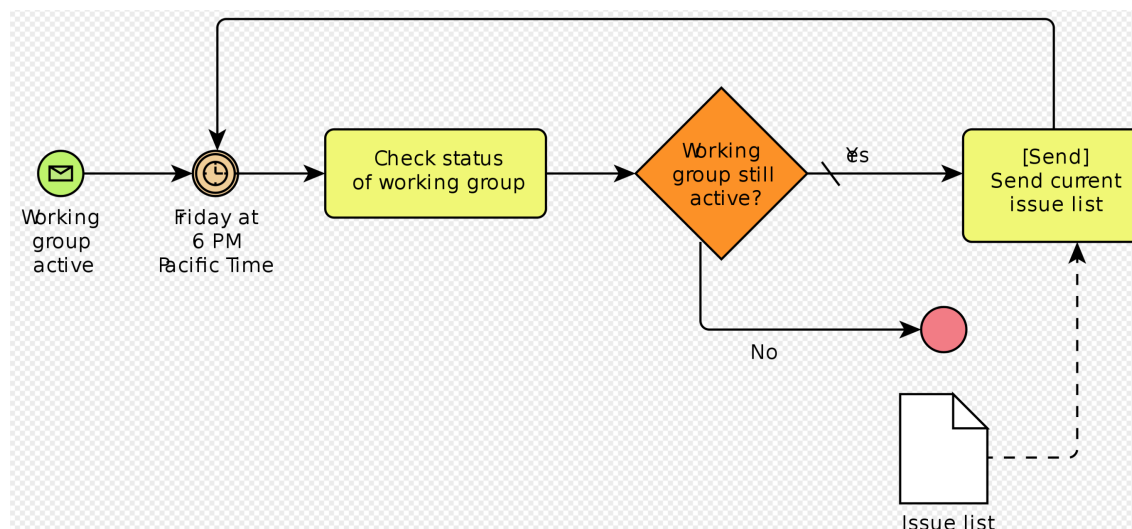
Types of system modeling

In business and IT development systems are modeled with different scopes and scales of complexity, such as:

- [Functional modeling](#)
- [Systems architecture](#)
- [Business process modeling](#)
- [Enterprise modeling](#)

(Source: https://en.wikipedia.org/wiki/Systems_modeling)

Business Process Model and Notation (BPMN) is a graphical representation for specifying business processes in a workflow. BPMN was developed by Business Process Management Initiative (BPMI), and is currently maintained by the Object Management Group since the two organizations merged in 2005. The current version of BPMN is 2.0.^[29]

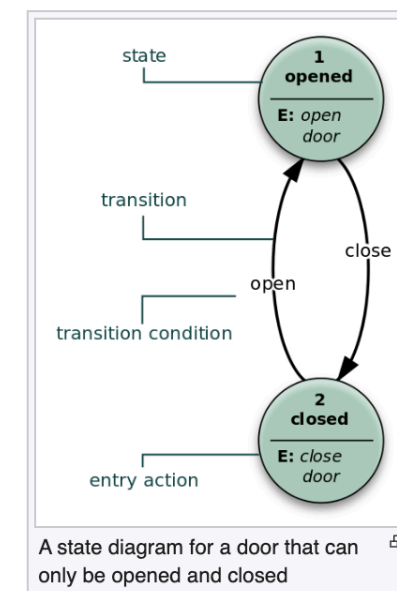


Specific types of modeling languages^[edit]

- [Framework-specific modeling language](#)
- [Systems Modeling Language](#)

SysML includes 9 types of diagram, some of which are taken from UML.

- [Block definition diagram](#)
- [Internal block diagram](#)
- [Package diagram](#)
- [Use case diagram](#)
- [Requirement Diagram](#)
- [Activity diagram](#)
- [Sequence diagram](#)
- [State machine diagram](#)
- [Parametric diagram](#)














Workflow engine vs. Statemachine

Category	Workflow Engine	Statemachine
Flexibility	sequential pattern	unordered
Understandability	complex and appropriate for large systems	easy but complicated to implement in large systems
Readability	easy to read in large systems	hard to read in large systems
Predictability	predictable and predefined	randomly as it depends on external events
When to use?	complex systems	small systems
Standard	BPMN 2.x	None
Graphical representation	BPMN components	Represented through UML state diagrams

Source: <https://workflowengine.io/blog/workflow-engine-vs-state-machine/>

Building blocks of Statemachine

Item	Description	UML Representation
State	model of the system	
Transition	relationship between a source state and a target state	
Internal transition	the source state and the target state are always the same	
Event	the impulse/signal which triggers a state change	
Initial state	the initial state of the machine(mandatory)	 Initial
End state	final state(mandatory)	 FinalState
History state	pseudo state linking previous state	 ShallowHistory
Choice state	pseudo state allowing a transition choice based on some input (similar to if/elseif/else	 Choice
Junction state	similar to choice state but has multiple incoming transitions	 Junction
Region	composite state or sub state machine	 Region
Action	behavior run during state transition	usually specified on the transition arrow
Guard	an expression evaluated dynamically based on variables and parameters	
Entry point	pseudo state when entering into submachine	 EntryPoint
Exit point	pseudo state when exiting submachine	 ExitPoint

What?

SSM:  **spring** by Pivotal

- ▶ Implements the traditional state machine
- ▶ Integration with Spring IoC and Spring Boot
- ▶ Hierarchical state machine structure
- ▶ Support for regions, sub-machines
- ▶ Support for Eclipse modeling framework (Papyrus)
- ▶ Supports distributed states through the use of Zookeeper
- ▶ Testing support
- ▶ 3rd party modules: <https://docs.spring.io/spring-statemachine/docs/3.0.0.M1/reference/#modules>

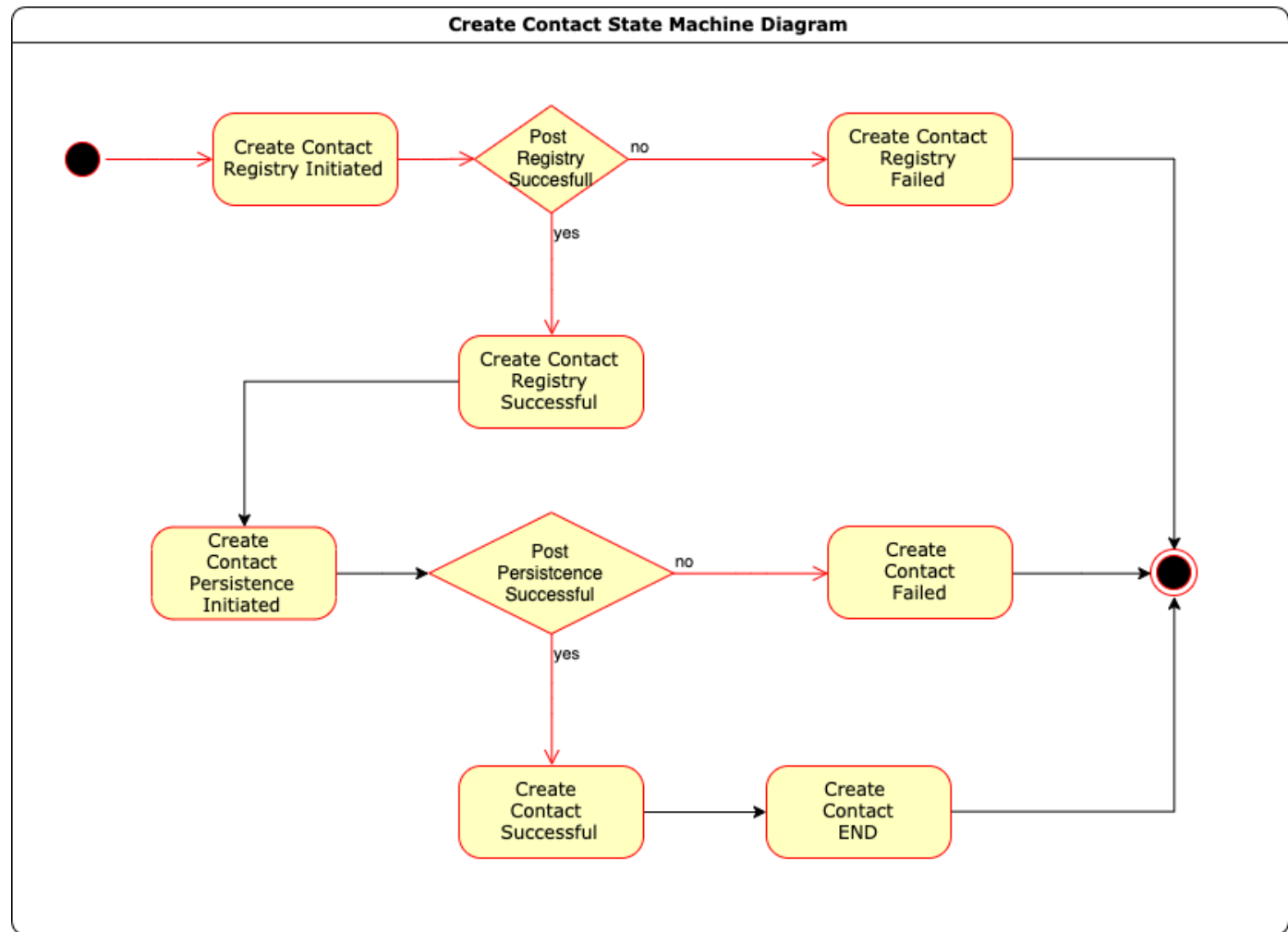
Why?

Traditionally, state machines are added to an existing project when developers realize that the code base is starting to look like a plate full of spaghetti. Spaghetti code looks like a never ending, hierarchical structure of IF, ELSE, and BREAK clauses, and compilers should probably ask developers to go home when things are starting to look too complex.

You are already trying to implement a state machine when you:

- Use boolean flags or enums to model situations.
- Have variables that have meaning only for some part of your application lifecycle.
- Loop through an if-else structure (or, worse, multiple such structures), check whether a particular flag or enum is set, and then make further exceptions about what to do when certain combinations of your flags and enums exist or do not exist.

Hands-on: Modeling RESTfull APIs using SSM



Create Contact

```

public void configure(StateMachineTransitionConfigurer<CreateContactState, CreateContactEvent> transitions)
    throws Exception {
    // @formatter:off
    transitions
        .withExternal()
            .source(START).target(CONTACT_REGISTRY_INITIATED)
            .event(CreateContactEvent.START)
            .action(createRegistryAction::contactRegistryInitiated)
            .action(createRegistryAction::postContact)
            .action(createRegistryAction::sendContactRegistryInitiated)
            .and()

        .withExternal()
            .source(CONTACT_REGISTRY_INITIATED).target(CONTACT_REGISTRY_CHOICE)
            .event(CreateContactEvent.CONTACT_REGISTRY_INITIATED)
            .action(createRegistryAction::contactRegistrySuccess)
            .and()

        .withChoice()
            .source(CONTACT_REGISTRY_CHOICE)
            .first(CONTACT_REGISTRY_SUCCESS, createRegistryChoice())
            .last(CONTACT_REGISTRY_ERROR)
            .and()

        .withExternal()
            .source(CONTACT_REGISTRY_SUCCESS).target(CONTACT_PERSISTENCE_INITIATED)
            .event(CreateContactEvent.CONTACT_PERSISTENCE_INITIATED)
            .action(createPersistenceAction::contactPersistenceInitiated)
            .and()

        .withExternal()
            .source(CONTACT_PERSISTENCE_INITIATED).target(CONTACT_PERSISTENCE_CHOICE)
            .event(CreateContactEvent.CONTACT_PERSISTENCE_INITIATED)
            .action(createPersistenceAction::contactPersistenceSuccess)
            .and()

        .withChoice()
            .source(CONTACT_PERSISTENCE_CHOICE)
            .first(CONTACT_PERSISTENCE_SUCCESS, createPersistenceChoice())
            .last(CONTACT_PERSISTENCE_ERROR)
            .and()

        .withExternal()
            .source(CONTACT_PERSISTENCE_SUCCESS).target(END)
            .event(CreateContactEvent.STOP)
            .action(createAction::createContactEnd);
    // @formatter:on
}
    
```