# Модуль Common.utilities

package tu.coreservice.utilities

import com.typesafe.config.{Config, ConfigFactory}

import org.slf4j.{Logger, LoggerFactory}

import tu.model.knowledge.Constant

/\*\*

\* Contains configuration for whole solution.

\* Use file tu.properties with string useProxy = true.

\* Put tu.properties in classpath.

\* @author toschev alex

\* Date: 28.05.12

\* Time: 20:35

\*/

object Configurator {

val log: Logger = LoggerFactory.getLogger(this.getClass)

/\*\*

\* Proxy description

\*/

def proxyAddress(): ProxyDescription = {

val res = new ProxyDescription

try {

val proxyCfg: Config = ConfigFactory.load(Constant.tuIniAddress)

res.proxyHost = proxyCfg.getString("proxyHost")

res.proxyPort = proxyCfg.getInt("proxyPort")

res.useProxy = proxyCfg.getBoolean("useProxy")

log.debug("Use proxy settings from config host {}:{} use: {}", List(res.proxyHost, res.proxyPort.toString, res.useProxy))

return res

}

catch {

case ex: Exception => log.debug(ex.getMessage)

}

res

}

def spellingAtDServer = "http://service.afterthedeadline.com/"

/\*\*

\* select spell checker engine GOOGLE, ATD or COMPOUND

\* ATD - after the deadline

\* COMPOUND - spell checker from GOOGLE and grammar checker from After The Deadline

\*/

def spellCheckerEngine(): String = "ATD"

}

package tu.coreservice.utilities

import tu.model.knowledge.Constant

/\*\*

\* Is multi culture support

\* @author Alexander Toschev

\* Date: 9/9/12

\* Time: 7:33 PM

\*/

object LocalizedResources {

private val localizationMap = Map("$ErrorOccured" -> "An error has occurred:",

"$ProvideAdditionalInfo" -> "Please provide additional information",

"$ClarifyPhrases" -> "Sorry, i can't understand next phrases: ",

"$ClarifyConcepts" -> "Sorry, i can't understand next concepts: ",

Constant.UNDERSTOOD\_CONCEPTS -> "Understood concepts: ",

Constant.NOT\_UNDERSTOOD\_CONCEPTS -> "Not understood concepts: ",

Constant.FOUND\_SOLUTIONS -> "Found solutions: "

)

/\*\*

\* return localized string or string ig nothing found.

\* @param key the string to localize.

\* @return localized string in target language.

\*/

def GetString(key: String): String = {

if (localizationMap.contains(key))

return localizationMap(key)

key

}

}

package tu.coreservice.utilities

import org.slf4j.Logger

import scala.collection.JavaConversions.\_

/\*\*

\* @author max

\* date 2012-11-11

\* time: 2:16 AM

\*/

object LoggerHelper {

def info(logger: Logger, message: String, objects: List[AnyRef]) {

logger.info(message, objects.toArray)

}

}

package tu.coreservice.utilities

/\*\*

\* @author toschev alex

\* Date: 28.05.12

\* Time: 20:41

\*/

class ProxyDescription {

var proxyHost = ""

var proxyPort = 8080

var proxyType = "HTTP"

var useProxy = false

}

package tu.coreservice.utilities

import tu.model.knowledge.domain.{ConceptTag, ConceptLink, ConceptNetwork, Concept}

import tu.model.knowledge.annotator.{AnnotatedNarrative, AnnotatedPhrase}

import tu.model.knowledge.{KnowledgeURI, Probability}

import tu.model.knowledge.howto.HowTo

import tu.model.knowledge.frame.Frame

/\*\*

\* Test data generator object.

\* @author talanov max

\* date 2012-06-16

\* time: 3:55 PM

\*/

@deprecated object TestDataGenerator {

/\*\*

\* Simulation structures

\*/

val namespace = "2/concepts/"

val revision = "0.0"

/\*\*

\* concepts

\*/

val tenseConcept = Concept("tense")

val posConcept = Concept("pos")

val subjectConcept = Concept("subject")

val objectConcept = Concept("object")

val systemConcept = Concept.createSubConcept(objectConcept, "system")

val userConcept = Concept.createSubConcept(subjectConcept, "user")

val computerConcept = Concept.createSubConcept(objectConcept, "computer")

val deviceConcept = Concept.createSubConcept(objectConcept, "device")

val softwareConcept = Concept.createSubConcept(objectConcept, "sofware")

val versionConcept = Concept.createSubConcept(objectConcept, "version")

val photoShopConcept = Concept.createSubConcept(softwareConcept, "Adobe Photoshop")

val browserConcept = Concept.createSubConcept(softwareConcept, "Browser")

val firefoxConcept = Concept.createSubConcept(browserConcept, "Mozilla Firefox")

val internetExplorerConcept = Concept.createSubConcept(browserConcept, "Microsoft Internet Explorer")

val networkConcept = Concept.createSubConcept(objectConcept, "network")

val addressConcept = Concept.createSubConcept(objectConcept, "address")

val internetConcept = Concept.createSubConcept(networkConcept, "internet")

val sharedResourcesConcept = Concept.createSubConcept(objectConcept, "sharedResources")

val sharedDiskConcept = Concept.createSubConcept(sharedResourcesConcept, "sharedDisk")

val accountConcept = Concept.createSubConcept(objectConcept, "account")

// axillary

val formOfPoliteness = Concept("formOfPoliteness")

val pls=Concept.createSubConcept(formOfPoliteness,"please")

// actions

val actionConcept = Concept("action")

val installConcept = Concept.createSubConcept(actionConcept, "install")

val removeConcept = Concept.createSubConcept(actionConcept, "remove")

val cleanConcept = Concept.createSubConcept(actionConcept, "clean")

// problems

val problemConcept = Concept.createSubConcept(objectConcept, "problem")

val lackConcept = Concept.createSubConcept(problemConcept, "lack")

/\*\*

\* Desired states

\*/

val desireConcept = Concept("desire")

val shouldConcept = Concept.createSubConcept(desireConcept, "should")

val shouldHaveConcept = Concept.createSubConcept(shouldConcept, "shouldHave")

var concepts = List[Concept](systemConcept, subjectConcept, objectConcept, userConcept, computerConcept, softwareConcept,

photoShopConcept, browserConcept,

firefoxConcept, internetExplorerConcept,

networkConcept, internetConcept, sharedResourcesConcept,

formOfPoliteness,

sharedDiskConcept, accountConcept, actionConcept, versionConcept,

actionConcept, installConcept, removeConcept, cleanConcept)

var simulationConcepts = List[Concept](systemConcept, subjectConcept, objectConcept, userConcept, computerConcept, softwareConcept,

photoShopConcept, browserConcept,

firefoxConcept, internetExplorerConcept,

networkConcept, internetConcept, sharedResourcesConcept,

sharedDiskConcept, accountConcept, actionConcept, versionConcept,

actionConcept, installConcept, removeConcept, cleanConcept)

var reformulationConcepts = List[Concept](systemConcept, subjectConcept, objectConcept, userConcept, computerConcept, softwareConcept,

photoShopConcept, browserConcept,

firefoxConcept, internetExplorerConcept,

networkConcept, internetConcept, sharedResourcesConcept,

sharedDiskConcept, accountConcept, actionConcept, versionConcept,

actionConcept, installConcept, removeConcept, cleanConcept)

/\*\*

\* links

\*/

/\*\*

\* has

\*/

val has = ConceptLink(subjectConcept, objectConcept, "has")

val hasComputer = ConceptLink.createSubConceptLink(has, userConcept, computerConcept, "hasComputer", new Probability(1.0, 1.0))

val userComputerLinkedPair = ConceptLink.likConcepts(hasComputer, userConcept, computerConcept)

val hasSoftware = ConceptLink.createSubConceptLink(has, computerConcept, softwareConcept, "hasSoftware", new Probability(1.0, 0.9))

val computerSoftwareLinkedPair = ConceptLink.likConcepts(hasSoftware, computerConcept, softwareConcept)

val hasAccount = ConceptLink.createSubConceptLink(has, userConcept, accountConcept, "hasAccount", new Probability(1.0, 0.9))

val userAccountLinkedPair = ConceptLink.likConcepts(has, userConcept, accountConcept)

val hasVersion = ConceptLink.createSubConceptLink(has, softwareConcept, versionConcept, "hasVersion", new Probability(1.0, 0.9))

val hasVersionLinkedPair = ConceptLink.likConcepts(has, softwareConcept, versionConcept)

/\*\*

\* uses

\*/

val isUsedFor = ConceptLink(subjectConcept, objectConcept, "isUsedFor")

val browserIsUsedForInternet = ConceptLink.createSubConceptLink(isUsedFor, browserConcept, internetConcept, "browserIsUsedForInternet")

val isLink = ConceptLink(subjectConcept, objectConcept, "is")

val appliedLink = ConceptLink(subjectConcept, objectConcept, "applied")

val missLink = ConceptLink(userConcept, objectConcept, "miss")

val hasNo = ConceptLink(subjectConcept, objectConcept, "hasNo")

val hasProblemWith = ConceptLink(subjectConcept, objectConcept, "hasProblemWith")

val tenseLink = ConceptLink(subjectConcept, objectConcept, "tense")

val posLink = ConceptLink(subjectConcept, objectConcept, "pos")

// actions

val actionLink = ConceptLink(subjectConcept, objectConcept, "actionLink")

val installLink = ConceptLink.createSubConceptLink(actionLink, subjectConcept, softwareConcept, "install")

val removeConceptLink = ConceptLink.createSubConceptLink(actionLink, subjectConcept, softwareConcept, "remove")

val cleanConceptLink = ConceptLink.createSubConceptLink(actionLink, subjectConcept, deviceConcept, "clean")

var conceptLinks: List[ConceptLink] = List(has, hasComputer, hasSoftware, hasAccount, hasVersion, isLink, appliedLink,

actionLink, installLink, cleanConceptLink, removeConceptLink, cleanConceptLink, tenseLink, posLink,missLink)

var simulationConceptLinks: List[ConceptLink] = List(has, hasComputer, hasSoftware, hasAccount, hasVersion, isLink, appliedLink,

actionLink, installLink, cleanConceptLink, removeConceptLink, cleanConceptLink)

var reformulationConceptLinks: List[ConceptLink] = List(has, hasComputer, hasSoftware, hasAccount, hasVersion, isLink, appliedLink,

actionLink, installLink, cleanConceptLink, removeConceptLink, cleanConceptLink)

/\*\*

\* Domain model concept network

\*/

val domainModel = ConceptNetwork(concepts, conceptLinks, KnowledgeURI("domainModel"))

/\*\*

\* Simulation model concept network

\*/

val simulationModel = ConceptNetwork(simulationConcepts, simulationConceptLinks, KnowledgeURI("simulationModel"))

/\*\*

\* Reformulation model concept network

\*/

val reformulationModel = ConceptNetwork(reformulationConcepts, reformulationConceptLinks, KnowledgeURI("simulationModel"))

/\*\*

\* sentences

\*/

val userPhrase = AnnotatedPhrase("user", userConcept)

val customerPhrase = AnnotatedPhrase("customer", userConcept)

val notebookPhrase = AnnotatedPhrase("notebook", computerConcept)

val laptopPhrase = AnnotatedPhrase("laptop", computerConcept)

val desktopPhrase = AnnotatedPhrase("desktop", computerConcept)

val computerPhrase = AnnotatedPhrase("computer", computerConcept)

/\*\*

\* HowTo-s

\*/

val installHowTo = new HowTo(List[Frame](Frame(objectConcept)), List[ConceptTag](), KnowledgeURI("installHowTo"))

val reinstallHowTo = new HowTo(List[Frame](Frame(objectConcept)), List[ConceptTag](), KnowledgeURI("reinstallHowTo"))

/\*\*

\* Test sentences

\*/

// Please install Firefox

val please = AnnotatedPhrase("Please")

val install = AnnotatedPhrase("install")

val fireFox = AnnotatedPhrase("FireFox")

val pleaseInstallFF = AnnotatedNarrative(List(please, install, fireFox), KnowledgeURI("pleaseInstallFF"))

// Please install Firefox annotated

val pleaseAnnotatedPhrase = AnnotatedPhrase("Please", Concept("please"))

val installAnnotatedPhrase = AnnotatedPhrase("install", installConcept)

val fireFoxAnnotatedPhrase = AnnotatedPhrase("FireFox", firefoxConcept)

val pleaseInstallFFAnnotated = AnnotatedNarrative(List(pleaseAnnotatedPhrase, installAnnotatedPhrase, fireFoxAnnotatedPhrase),

KnowledgeURI("pleaseInstallFF"))

// Please install Firefox simulation

val installActionInst = Concept.createInstanceConcept(installConcept)

val firefoxConceptInst = Concept.createInstanceConcept(firefoxConcept)

val systemInstallFirefox = ConceptLink.createSubConceptLink(appliedLink, systemConcept, firefoxConceptInst, "systemInstallFirefox")

val pleaseInstallFFSimulation = new ConceptNetwork(List[Concept](installActionInst, firefoxConceptInst), List[ConceptLink](systemInstallFirefox),

KnowledgeURI("pleaseInstallFFSimulation"))

// User miss Internet Explorer 8

val user = AnnotatedPhrase("User")

val missing = AnnotatedPhrase("miss")

val internetExplorer8 = AnnotatedPhrase("Internet Explorer 8")

val iHaveProblemWithIE8 = AnnotatedNarrative(List(user, missing, internetExplorer8), KnowledgeURI("iHaveProblemWithIE8"))

// User miss Internet Explorer 8 annotated

val userAnnotatedPhrase = AnnotatedPhrase("User", userConcept)

val missingAnnotatedPhrase = AnnotatedPhrase("miss", lackConcept)

val internetExplorer8AnnotatedPhrase = AnnotatedPhrase("Internet Explorer", internetExplorerConcept)

val iHaveProblemWithIE8Annotated = AnnotatedNarrative(List(userAnnotatedPhrase,

missingAnnotatedPhrase, internetExplorer8AnnotatedPhrase), KnowledgeURI("iHaveProblemWithIE8"))

// User miss Internet Explorer 8 annotated

val shouldHaveAnnotatedPhrase = AnnotatedPhrase("should have", shouldHaveConcept)

val iHaveProblemWithIE8IShouldHaveIE8 = AnnotatedNarrative(List(userAnnotatedPhrase,

missingAnnotatedPhrase, internetExplorer8AnnotatedPhrase,

shouldHaveAnnotatedPhrase, internetExplorer8AnnotatedPhrase), KnowledgeURI("iHaveProblemWithIE8IShouldHaveIE8"))

// User miss Internet Explorer 8 annotated

val lackLink = ConceptLink(lackConcept, internetExplorerConcept, "lackLink")

val internetExplorer8AnnotatedPhraseAmbiguous = AnnotatedPhrase("Internet Explorer 8", List(objectConcept, internetExplorerConcept))

val iHaveProblemWithIE8AnnotatedAmbiguous = AnnotatedNarrative(List(userAnnotatedPhrase,

missingAnnotatedPhrase, internetExplorer8AnnotatedPhraseAmbiguous), KnowledgeURI("iHaveProblemWithIE8"))

val phrases: List[AnnotatedPhrase] = pleaseInstallFFAnnotated.sentences.map(s => s.phrases).flatten ::: iHaveProblemWithIE8IShouldHaveIE8.sentences.map(s => s.phrases).flatten

// User miss Internet Explorer 8 simulated

val userInst = Concept.createInstanceConcept(userConcept)

val internetExplorerInst = Concept.createInstanceConcept(internetExplorerConcept)

val versionInst = Concept.createInstanceConcept(versionConcept, "8")

val userMissInternetExplorer = ConceptLink.createInstanceConceptLink(missLink, userInst, internetExplorerInst)

val internetExplorerHasVersion = ConceptLink.createInstanceConceptLink(has, internetExplorerInst, versionInst)

val iHaveProblemWithIE8Simulation = new ConceptNetwork(List[Concept](userInst, internetExplorerInst, versionInst),

List[ConceptLink](userMissInternetExplorer, internetExplorerHasVersion), KnowledgeURI("iHaveProblemWithIE8Simulation"))

// User miss Internet Explorer 8 reformulated

val userInstRef = Concept.createInstanceConcept(userConcept)

val computerInstRef = Concept.createInstanceConcept(computerConcept)

val userHasComputerInst = ConceptLink.createInstanceConceptLink(has, userInstRef, computerInstRef)

val addressInstRef = Concept.createInstanceConcept(addressConcept, "someAddress")

val computerHasAddressRef = ConceptLink.createInstanceConceptLink(has, computerInstRef, addressInstRef)

val internetExplorerInstRef = Concept.createInstanceConcept(internetExplorerConcept)

val computerHasNoInternetExplorerInstRef = ConceptLink.createInstanceConceptLink(hasNo, computerInstRef, internetExplorerInstRef)

val internetExplorerHasVersionInstRef = ConceptLink.createInstanceConceptLink(has, internetExplorerInstRef, versionInst)

val iHaveProblemWithIE8Reformulation = new ConceptNetwork(List[Concept](userInstRef, computerInstRef, addressInstRef, internetExplorerInstRef),

List[ConceptLink](userHasComputerInst, computerHasAddressRef, computerHasNoInternetExplorerInstRef, internetExplorerHasVersionInstRef), KnowledgeURI("iHaveProblemWithIE8Reformulation"))

val iHaveProblemWithIE8ReformulationTest = new ConceptNetwork(List[Concept](userConcept, computerConcept, addressConcept, internetExplorerConcept),

List[ConceptLink](), KnowledgeURI("iHaveProblemWithIE8Reformulation"))

val installFirefoxHowTo = HowTo.createInstance(installHowTo, List(Frame(firefoxConcept)))

val reinstallIEHowTo = HowTo.createInstance(installHowTo, List(Frame(internetExplorerConcept)))

def generateDirectInstructionNarrative = pleaseInstallFF

def generateProblemDescriptionNarrative = iHaveProblemWithIE8

def generateDomainModelConceptNetwork = domainModel

def generateSimulationModelConceptNetwork = simulationModel

def generateReformulationModelConceptNetwork = reformulationModel

/\*\*

\* Generates AnnotatedNarrative the result of KBAnnotator.

\* @return AnnotatedNarrative

\*/

def generateDirectInstructionAnnotatedNarrative = pleaseInstallFFAnnotated

/\*\*

\* Generates AnnotatedNarrative the result of KBAnnotator.

\* @return AnnotatedNarrative

\*/

def generateProblemDescriptionAnnotatedNarrative = iHaveProblemWithIE8Annotated

/\*\*

\* Generates AnnotatedNarrative with desired state the result of KBAnnotator.

\* @return AnnotatedNarrative

\*/

def generateProblemDescriptionWithDesiredStateAnnotatedNarrative = iHaveProblemWithIE8Annotated

/\*\*

\* Generates AnnotatedNarrative with ambiguous sentences(that references several concepts)

\* @return

\*/

def generateProblemDescriptionAnnotatedNarrativeAmbiguous = iHaveProblemWithIE8AnnotatedAmbiguous

/\*\*

\* Generates ConceptNetwork the result of Simulation.

\* @return ConceptNetwork

\*/

def generateDirectInstructionSimulation = pleaseInstallFFSimulation

/\*\*

\* Generates ConceptNetwork the result of Simulation.

\* @return ConceptNetwork

\*/

def generateProblemDescriptionSimulation = iHaveProblemWithIE8Simulation

/\*\*

\* Generates ConceptNetwork the result of Reformulation.

\* @return ConceptNetwork

\*/

def generateProblemDescriptionReformulation = iHaveProblemWithIE8Reformulation

/\*\*

\* Generates Test ConceptNetwork for Reformulation.

\* @return ConceptNetwork.

\*/

def generateProblemDescriptionReformulationTest = iHaveProblemWithIE8ReformulationTest

/\*\*

\* Generates install Firefox HowTo

\* @return HowTo

\*/

def generateInstallFirefoxHowTo = installFirefoxHowTo

/\*\*

\* Generates reinstall IE8 HowTo

\* @return HowTo

\*/

def generateReinstallIE8HowTo = reinstallIEHowTo

}

package tu.coreservice.utilities

import java.net.URI

/\*\*

\* @author toscheva

\* Date: 06.06.12

\* Time: 10:50

\*

\*/

/\*\*

\* Contain helpers to work with URI

\*/

object URIHelper {

/\*\*

\* Check if URI contain specific tag

\* @param tag tagName

\* @param uri target URI

\* @return true if uri contains tag

\*/

def isContainTag(tag: String, uri: URI): Boolean = {

uri.toString.contains(tag)

}

/\*\*

\* generate empty uri

\* @return empty uri

\*/

def emptyURI() = "http://tu-project.com/knowledge/empty"

def uriPrefix() = "http://tu-project.com/"

def uriProjectName = "tu-project.com"

def generateURIForFrame(frameUniqueName: String): String = {

"http://tu-project.com/" + frameUniqueName

}

/\*\*

\*

\* @return namespace for annotator

\*/

def annotatorNamespace(): String = {

"tu.coreservice.annotator"

}

/\*\*

\* return project version

\* @return project version

\*/

def version(): String = {

"1.0"

}

/\*\*

\* text mark for splitter results

\* @return

\*/

def splitterMark(): String = {

// return "splitted"

"tu.coreservice.splitter.PreliminarySplitter"

}

def sentenceMark(): String = {

"sentense"

}

}

# Модуль coreservice.action.critic

package tu.coreservice.action.critic

import tu.coreservice.action.Action

import tu.model.knowledge.{Probability, KnowledgeURI}

import tu.model.knowledge.communication.ShortTermMemory

/\*\*

\* Critic trait.

\* @author toschev alex

\* @author talanov max

\* Date: 03.05.12

\*/

abstract class Critic(\_excluded: List[CriticLink], \_included: List[CriticLink], \_uri: KnowledgeURI, \_probability: Probability)

extends Action(\_uri, \_probability) {

/\*\*

\* Returns excluded by current CriticLink-s, if(current) => (!excluded)

\* @return List[Critic] excluded critics.

\*/

def exclude(): List[CriticLink] = \_excluded

/\*\*

\* Returns included in current CriticLink-s, if(current) => (included).

\* @return List[Critic] included critics.

\*/

def include(): List[CriticLink] = \_included

/\*\*

\* Generic method of the action to be applied over input ShortTermMemory and put all results in output ShortTermMemory.

\* @param inputContext ShortTermMemory of all inbound parameters

\* @return output ShortTermMemory.

\*/

def apply(inputContext: ShortTermMemory): ShortTermMemory

}

package tu.coreservice.action.critic

import tu.model.knowledge.{KnowledgeURI, Link}

/\*\*

\* @author max talanov

\* date 2012-07-01

\* time: 4:47 PM

\*/

class CriticLink(\_source: Critic, \_destination: Critic, \_uri: KnowledgeURI)

extends Link[Critic](\_source, \_destination, \_uri) {

}

package tu.coreservice.action.critic.analyser

import tu.coreservice.action.critic.{CriticLink, Critic}

import tu.model.knowledge.{Resource, Probability, KnowledgeURI}

import tu.model.knowledge.domain.ConceptNetwork

import tu.model.knowledge.selector.SelectorRequest

import tu.model.knowledge.communication.{ContextHelper, ShortTermMemory}

import tu.model.knowledge.annotator.AnnotatedNarrative

import tu.coreservice.action.way2think.cry4help.Cry4HelpWay2Think

/\*\*

\* @author max talanov

\* date 2012-10-04

\* time: 12:19 AM

\*/

abstract class Analyser(\_exclude: List[CriticLink], \_include: List[CriticLink], \_uri: KnowledgeURI, \_probability: Probability = new Probability())

extends Critic(\_exclude, \_include, \_uri, \_probability) {

/\*\*

\* Estimates confidence and probability of output SelectorRequest

\* @param currentSituation description of current situation as ConceptNetwork

\* @param domainModel overall domain model to be used to analyse current situation as ConceptNetwork.

\* @return SelectorRequest with set probability

\*/

def apply(currentSituation: ConceptNetwork, domainModel: ConceptNetwork): SelectorRequest

override def apply(inputContext: ShortTermMemory): ShortTermMemory = {

// get lastResult ConceptNetwork from inputContext

try {

inputContext.lastResult match {

case Some(narrative: AnnotatedNarrative) => {

inputContext.domainModel match {

case Some(domainModel: ConceptNetwork) => {

val selectorRequest = this.apply(narrative.conceptNetwork, domainModel)

ContextHelper(List[Resource](), selectorRequest, this.getClass.getName + " result")

}

case None => {

val cry4Help = Cry4HelpWay2Think("$No\_domain\_model\_specified")

// throw new UnexpectedException("$No\_domain\_model\_specified")

ContextHelper(List[Resource](cry4Help), cry4Help, this.getClass.getName + " result")

}

}

}

case None => {

val cry4Help = Cry4HelpWay2Think("$Context\_lastResult\_is\_None")

// throw new UnexpectedException("$Context\_lastResult\_is\_not\_expectedType " + e.getMessage)

ContextHelper(List[Resource](cry4Help), cry4Help, this.getClass.getName + " result")

}

}

} catch {

case e: ClassCastException => {

val cry4Help = Cry4HelpWay2Think("$Context\_lastResult\_is\_not\_expected\_type " + e.getMessage)

// throw new UnexpectedException("$Context\_lastResult\_is\_not\_expectedType " + e.getMessage)

ContextHelper(List[Resource](cry4Help), cry4Help, this.getClass.getName + " result")

}

}

}

}

package tu.coreservice.action.critic.analyser

import tu.model.knowledge.domain.{ConceptLink, Concept, ConceptNetwork}

import tu.model.knowledge.selector.SelectorRequest

import tu.model.knowledge.{Constant, Probability, KnowledgeURI}

import tu.model.knowledge.annotator.AnnotatedNarrative

import org.slf4j.LoggerFactory

/\*\*

\* Direct Instruction Analyser, detects direct instruction and return Selector request with proper Probability.

\* @author max talanov

\* date 2012-06-28

\* time: 4:01 PM

\*/

class DirectInstructionAnalyser {

val log = LoggerFactory.getLogger(this.getClass)

/\*\*

\* Detects direct instruction and return SelectorRequest with proper Probability.

\* @param currentSituation AnnotatedNarrative to analyse. \

\* @return SelectorRequest with calculated probability.

\*/

def apply(currentSituation: AnnotatedNarrative): SelectorRequest = {

this.apply(currentSituation.concepts)

}

/\*\*

\* Detects direct instruction and return SelectorRequest with proper Probability.

\* @param currentSituation ConceptNetwork to analyse.

\* @return SelectorRequest with calculated probability.

\*/

def apply(currentSituation: ConceptNetwork): SelectorRequest = {

this.apply(currentSituation.nodes)

}

/\*\*

\* Detects direct instruction and return SelectorRequest with proper Probability.

\* @param currentSituation List[Concept] to analyse.

\* @return SelectorRequest with calculated probability.

\*/

def apply(currentSituation: List[Concept]): SelectorRequest = {

var frequencyConfidence: Pair[Double, Double] = (1.0, 1.0)

// current situation must be either request to the system or impersonal sentence

val subjects = ConceptNetwork.getNodeByGeneralisationName(currentSituation, Constant.SUBJECT\_NAME)

frequencyConfidence = if (subjects.size > 1) {

(frequencyConfidence.\_1 - 1.0, 1.0)

} else if (subjects.size == 1 && subjects(0).uri.name != Constant.SYSTEM\_NAME) {

(frequencyConfidence.\_1 - 1.0, 1.0)

} else {

(frequencyConfidence.\_1 - 0.0, 1.0)

}

//if current situation contains action and has link to an object to be applied

if (frequencyConfidence.\_1 > 0) {

val actions = ConceptNetwork.getNodeByGeneralisationName(currentSituation, Constant.ACTION\_NAME)

frequencyConfidence = if (actions.size == 1) {

(frequencyConfidence.\_1 - 0.0, 0.9)

} else if (actions.size > 1) {

(frequencyConfidence.\_1 - 0.5, 0.9)

} else {

(frequencyConfidence.\_1 - 1.0, 1.9)

}

// if action has object

if (frequencyConfidence.\_1 > 0) {

val filteredActions = actions.filter {

concept: Concept => {

concept.links.filter {

link: ConceptLink => {

if (link.source != concept) {

link.destination != null

} else {

link.source != null

}

}

}.size > 0

}

}

if (filteredActions.size > 0) {

(frequencyConfidence.\_1 - 0.0, 0.9)

} else {

(frequencyConfidence.\_1 - 1.0, 1.0)

}

}

}

if (frequencyConfidence.\_1 < 0) {

frequencyConfidence = (0.0, frequencyConfidence.\_2)

}

val res = new SelectorRequest(List(KnowledgeURI(Constant.SELECTOR\_REQUEST\_SIMULATION\_URI)), KnowledgeURI(Constant.SELECTOR\_REQUEST\_SIMULATION\_URI\_NAME),

new Probability(frequencyConfidence.\_1, frequencyConfidence.\_2))

log info("created selector request={}", res)

res

}

/\*\*

\* Start method used by ThinkingLifeCycle, not implemented currently.

\* @return true if succeed, otherwise false.

\* @version 2.0

\*/

def start() = false

/\*\*

\* Stop method used by ThinkingLifeCycle, not implemented currently.

\* @return true if succeed, otherwise false.

\* @version 2.0

\*/

def stop() = false

}

object DirectInstructionAnalyser {

def apply(name: String): DirectInstructionAnalyser = {

new DirectInstructionAnalyser()

}

}

package tu.coreservice.action.critic.analyser

import tu.coreservice.action.critic.{CriticLink, Critic}

import tu.model.knowledge.{Probability, KnowledgeURI}

import tu.model.knowledge.domain.ConceptNetwork

import tu.model.knowledge.selector.SelectorRequest

/\*\*

\* Critic adapter DirectInstructionAnalyser.

\* @author max talanov

\* date 2012-07-01

\* time: 6:45 PM

\*/

class DirectInstructionAnalyserCritic(\_exclude: List[CriticLink], \_include: List[CriticLink], \_uri: KnowledgeURI,

\_probability: Probability = new Probability())

extends Analyser(\_exclude, \_include, \_uri, \_probability) {

def this() = this(List[CriticLink](), List[CriticLink](), KnowledgeURI("DirectInstructionAnalyserCritic"))

def start() = false

def stop() = false

/\*\*

\* Estimates confidence and probability of output SelectorRequest

\* @param currentSituation description of current situation as ConceptNetwork

\* @param domainModel overall domain model to be used to analyse current situation as ConceptNetwork.

\* @return SelectorRequest with set probability

\*/

def apply(currentSituation: ConceptNetwork, domainModel: ConceptNetwork): SelectorRequest = {

val dIA = new DirectInstructionAnalyser()

val selectorRequest: SelectorRequest = dIA(currentSituation)

selectorRequest

}

}

package tu.coreservice.action.critic.analyser

import tu.model.knowledge.domain.{ConceptNetwork, Concept}

import tu.model.knowledge.selector.SelectorRequest

import tu.model.knowledge.{Probability, Constant, KnowledgeURI}

import tu.model.knowledge.annotator.AnnotatedNarrative

import org.slf4j.LoggerFactory

/\*\*

\* Problem Description Analyser, detects problem description and return Selector request with proper Probability.

\* @author max talanov

\* date 2012-07-02

\* time: 12:19 PM

\*/

class ProblemDescriptionAnalyser {

val log = LoggerFactory.getLogger(this.getClass)

/\*\*

\* Detects problem description and return SelectorRequest with proper Probability.

\* @param currentSituation AnnotatedNarrative to analyse.

\* @return SelectorRequest with calculated probability.

\*/

def apply(currentSituation: AnnotatedNarrative): SelectorRequest = {

this.apply(currentSituation.concepts)

}

/\*\*

\* Detects problem description and return SelectorRequest with proper Probability.

\* @param currentSituation ConceptNetwork to analyse.

\* @return SelectorRequest with calculated probability.

\*/

def apply(currentSituation: ConceptNetwork): SelectorRequest = {

this.apply(currentSituation.nodes)

}

/\*\*

\* Detects problem description and return SelectorRequest with proper Probability.

\* @param currentSituation List[Concept] to analyse.

\* @return SelectorRequest with calculated probability.

\*/

def apply(currentSituation: List[Concept]): SelectorRequest = {

var frequencyConfidence: Pair[Double, Double] = (1.0, 1.0)

// current situation must have at least one subject and it must not be System name.

val subjects = ConceptNetwork.getNodeByGeneralisationName(currentSituation, Constant.SUBJECT\_NAME)

frequencyConfidence = if (subjects.size >= 1) {

(frequencyConfidence.\_1 - 0.0, 1.0)

} else if (subjects.size == 1 && subjects(0).uri.name != Constant.SYSTEM\_NAME) {

(frequencyConfidence.\_1 - 0.0, 1.0)

} else {

(frequencyConfidence.\_1 - 1.0, 1.0)

}

if (frequencyConfidence.\_1 > 0) {

val actions = ConceptNetwork.getNodeByGeneralisationNameRec(currentSituation, Constant.PROBLEM\_NAME)

frequencyConfidence = if (actions.size == 1) {

(frequencyConfidence.\_1 - 0.0, 0.9)

} else if (actions.size > 1) {

(frequencyConfidence.\_1 - 0.5, 0.9)

} else {

(frequencyConfidence.\_1 - 1.0, 1.9)

}

}

val res = new SelectorRequest(

List(KnowledgeURI(Constant.SELECTOR\_REQUEST\_SIMULATION\_URI),

KnowledgeURI(Constant.SELECTOR\_REQUEST\_REFORMULATION\_URI)),

KnowledgeURI(Constant.SELECTOR\_REQUEST\_PROBLEM\_DESCRIPTION\_URI\_NAME),

new Probability(frequencyConfidence.\_1, frequencyConfidence.\_2))

log info("created selector request={}", res)

res

}

}

package tu.coreservice.action.critic.analyser

import tu.model.knowledge.{Probability, KnowledgeURI}

import tu.coreservice.action.critic.{CriticLink, Critic}

import tu.model.knowledge.domain.ConceptNetwork

import tu.model.knowledge.selector.SelectorRequest

/\*\*

\* ProblemDescriptionAnalyserCritic Action adapter to ProblemDescriptionAnalyser

\* @author max talanov

\* date 2012-07-10

\* time: 12:17 PM

\*/

class ProblemDescriptionAnalyserCritic(\_exclude: List[CriticLink], \_include: List[CriticLink], \_uri: KnowledgeURI, \_probability: Probability = new Probability())

extends Analyser(\_exclude, \_include, \_uri, \_probability) {

def this() = this(List[CriticLink](), List[CriticLink](), KnowledgeURI("ProblemDescriptionAnalyserCritic"))

def start() = false

def stop() = false

/\*\*

\* Estimates confidence and probability of output SelectorRequest

\* @param currentSituation description of current situation as ConceptNetwork

\* @param domainModel overall domain model to be used to analyse current situation as ConceptNetwork.

\* @return SelectorRequest with set probability

\*/

def apply(currentSituation: ConceptNetwork, domainModel: ConceptNetwork): SelectorRequest = {

val a = new ProblemDescriptionAnalyser()

val selectorRequest: SelectorRequest = a.apply(currentSituation)

selectorRequest

}

}

package tu.coreservice.action.critic.analyser

import tu.model.knowledge.domain.{ConceptNetwork, Concept}

import tu.model.knowledge.selector.SelectorRequest

import tu.model.knowledge.{Probability, KnowledgeURI, Constant}

import tu.model.knowledge.annotator.AnnotatedNarrative

import org.slf4j.LoggerFactory

/\*\*

\* Detects problem description with desired state and return SelectorRequest with proper Probability

\* @author max talanov

\* date 2012-07-02

\* time: 5:22 PM

\*/

class ProblemDescriptionWithDesiredStateAnalyser {

val log = LoggerFactory.getLogger(this.getClass)

/\*\*

\* Detects problem description with desired state and return SelectorRequest with proper Probability.

\* @param currentSituation AnnotatedNarrative to analyse.

\* @return SelectorRequest with calculated probability.

\*/

def apply(currentSituation: AnnotatedNarrative): SelectorRequest = {

this.apply(currentSituation.concepts)

}

/\*\*

\* Detects problem description with desired state and return SelectorRequest with proper Probability.

\* @param currentSituation ConceptNetwork to analyse.

\* @return SelectorRequest with calculated probability.

\*/

def apply(currentSituation: ConceptNetwork): SelectorRequest = {

this.apply(currentSituation.nodes)

}

/\*\*

\* Detects problem description with desired state and return SelectorRequest with proper Probability.

\* @param currentSituation List[Concept] to analyse.

\* @return SelectorRequest with calculated probability.

\*/

def apply(currentSituation: List[Concept]): SelectorRequest = {

var frequencyConfidence: Pair[Double, Double] = (1.0, 1.0)

// current situation must have at least one subject and it must not be System name.

val subjects = ConceptNetwork.getNodeByGeneralisationName(currentSituation, Constant.SUBJECT\_NAME)

frequencyConfidence = if (subjects.size > 1) {

(frequencyConfidence.\_1 - 0.0, 1.0)

} else if (subjects.size == 1 && subjects(0).uri.name != Constant.SYSTEM\_NAME) {

(frequencyConfidence.\_1 - 0.0, 1.0)

} else {

(frequencyConfidence.\_1 - 1.0, 1.0)

}

if (frequencyConfidence.\_1 > 0) {

val actions = ConceptNetwork.getNodeByGeneralisationNameRec(currentSituation, Constant.PROBLEM\_NAME)

frequencyConfidence = if (actions.size == 1) {

(frequencyConfidence.\_1 - 0.0, 0.9)

} else if (actions.size > 1) {

(frequencyConfidence.\_1 - 0.5, 0.9)

} else {

(frequencyConfidence.\_1 - 1.0, 1.9)

}

val desires = ConceptNetwork.getNodeByGeneralisationNameRec(currentSituation, Constant.DESIRE\_NAME)

frequencyConfidence = if (actions.size == 1) {

(frequencyConfidence.\_1 - 0.0, 0.9)

} else if (actions.size > 1) {

(frequencyConfidence.\_1 - 0.5, 0.9)

} else {

(frequencyConfidence.\_1 - 1.0, 1.9)

}

}

val res = new SelectorRequest(List(KnowledgeURI(Constant.SELECTOR\_REQUEST\_DIRECT\_INSTRUCTION\_URI\_NAME)),

KnowledgeURI(Constant.SELECTOR\_REQUEST\_PROBLEM\_DESCRIPTION\_URI\_NAME),

new Probability(frequencyConfidence.\_1, frequencyConfidence.\_2))

log info("created selector request={}", res)

res

}

}

package tu.coreservice.action.critic.analyser

import tu.model.knowledge.{Probability, KnowledgeURI}

import tu.coreservice.action.critic.{CriticLink, Critic}

import tu.model.knowledge.domain.ConceptNetwork

import tu.model.knowledge.selector.SelectorRequest

/\*\*

\* ProblemDescriptionWithDesiredStateAnalyserCritic Action interface to ProblemDescriptionWithDesiredStateAnalyser

\* @author max talanov

\* date 2012-07-10

\* time: 12:20 PM

\*/

class ProblemDescriptionWithDesiredStateAnalyserCritic(\_exclude: List[CriticLink], \_include: List[CriticLink], \_uri: KnowledgeURI, \_probability: Probability = new Probability())

extends Analyser(\_exclude, \_include, \_uri, \_probability) {

def this() = this(List[CriticLink](), List[CriticLink](), KnowledgeURI("DirectInstructionAnalyserCritic"))

def start() = false

def stop() = false

/\*\*

\* Estimates confidence and probability of output SelectorRequest

\* @param currentSituation description of current situation as ConceptNetwork

\* @param domainModel overall domain model to be used to analyse current situation as ConceptNetwork.

\* @return SelectorRequest with set probability

\*/

def apply(currentSituation: ConceptNetwork, domainModel: ConceptNetwork): SelectorRequest = {

val a = new ProblemDescriptionWithDesiredStateAnalyser()

val selectorRequest: SelectorRequest = a.apply(currentSituation)

selectorRequest

}

}

# Модуль coreservice.action.selector

package tu.coreservice.action.selector

import tu.model.knowledge.training.Goal

import tu.model.knowledge.{KnowledgeURI, Resource}

import tu.dataservice.knowledgebaseserver.KBAdapter

import tu.coreservice.action.way2think.cry4help.Cry4HelpWay2Think

import tu.model.knowledge.communication.Request

import tu.model.knowledge.selector.SelectorRequest

import tu.model.knowledge.domain.Concept

import tu.exception.UnexpectedStateException

/\*\*

\* @author max talanov

\* date 2012-07-05

\* time: 6:50 PM

\*/

class Selector {

def apply(goal: Goal): List[Resource] = {

//TODO get rid of KBAdapter

KBAdapter.getByGoalName(goal.uri.name) match {

case Some(resources) => {

resources

}

case None => {

// List(Cry4HelpWay2Think("$Can\_not\_find\_resources\_for\_goal"))

throw new UnexpectedStateException("Can not find resource for goal " + goal)

}

}

}

/\*

def apply(request: Request): List[Resource] = {

KBAdapter.goalResourceMap.get(Goal("ProcessIncident")) match {

case Some(resources) => {

resources

}

case None => {

List(Cry4HelpWay2Think("$Can\_not\_find\_resources\_for\_goal"))

}

}

}

\*/

def apply(request: SelectorRequest): List[Resource] = {

val resourcesOption: List[Option[Resource]] = request.resourceURIList.map {

uri: KnowledgeURI => KBAdapter.stringResourcesMap.get(uri.name)

}

val filteredResources: List[Option[Resource]] = resourcesOption.filter {

case Some(\_) => true

case None => false

}

val res: List[Resource] = filteredResources.map {

case Some(r: Resource) => r

}

res

}

//def workflow = KBAdapter.workflow

}

# Модуль coreservice.action.way2think

package tu.coreservice.action.way2think

import tu.model.knowledge.communication.{ContextHelper, ShortTermMemory}

import tu.model.knowledge.Resource

import tu.model.knowledge.selector.SelectorRequest

import org.slf4j.LoggerFactory

/\*\*

\* @author adel chepkunov, max Talanov, alexander toschev

\* Date: 30.06.12

\* Time: 9:12

\*/

class FindMostProbableAction extends Way2Think {

def start() = false

def stop() = false

/\*\*

\* Way2Think interface.

\* @param inputContext ShortTermMemory of all inbound parameters.

\* @return outputContext

\*/

def apply(inputContext: ShortTermMemory) = {

FindMostProbableAction(inputContext)

}

}

object FindMostProbableAction {

val log = LoggerFactory.getLogger(this.getClass)

def apply(inputContext: ShortTermMemory): ShortTermMemory = {

val outputContext = ContextHelper(List[Resource](), "OutputContex")

if (inputContext.classificationResults.isEmpty) {

outputContext.bestClassificationResult = None

outputContext.lastResult = None

}

else {

outputContext.classificationResults

= inputContext.classificationResults

.filter(p => !inputContext.checkedClassificationResults.exists(q => p == q))

.sortWith((s, t) => s.probability.frequency > t.probability.frequency)

outputContext.bestClassificationResult = outputContext.classificationResults.headOption

outputContext.lastResult = outputContext.classificationResults.headOption

log info("best classification result={}", outputContext.bestClassificationResult)

log info("last result={}", outputContext.lastResult)

outputContext.bestClassificationResult match {

case Some(sR: SelectorRequest) => {

log info("processed classification resluts={}", sR.toString)

outputContext.checkedClassificationResults = List(sR) ::: outputContext.checkedClassificationResults

}

case \_ => {

//none

}

}

outputContext.classificationResults = outputContext.classificationResults.tail

}

outputContext

}

}

package tu.coreservice.action.way2think

import tu.model.knowledge.communication.{ContextHelper, ShortTermMemory}

import tu.model.knowledge.domain.ConceptNetwork

import tu.model.knowledge.{Constant, Resource, SolvedIssue}

import org.slf4j.LoggerFactory

import tu.model.knowledge.narrative.Narrative

/\*\*

\* @author adel chepkunov

\* Date: 10.07.12

\* Time: 7:00

\*/

class SearchSolution extends Way2Think {

def start() = false

def stop() = false

/\*\*

\* Way2Think interface.

\* @param inputContext ShortTermMemory of all inbound parameters.

\* @return outputContext

\*/

def apply(inputContext: ShortTermMemory) = SearchSolution(inputContext)

}

object SearchSolution {

val log = LoggerFactory.getLogger(this.getClass)

val searcher = new Solutions

def apply(inputContext: ShortTermMemory): ShortTermMemory = {

searcher.solutions = inputContext.solutions

val res = inputContext.lastResult match {

case Some(cn: ConceptNetwork) => {

if (cn.rootNodes.size <= 0) {

return inputContext

}

searcher.search(cn, Nil)

}

case \_ => None

}

log debug("search solution result={}", res)

val outputContext = ContextHelper(List[Resource](), this.getClass.getName + " result")

outputContext.lastResult = res

this.setReport(res, outputContext)

outputContext

}

def setReport(solution: Option[SolvedIssue], context: ShortTermMemory): ShortTermMemory = {

solution match {

case Some(issue: SolvedIssue) => {

this.setResultsToReport(Constant.FOUND\_SOLUTIONS, context, List[SolvedIssue](issue))

}

case None => {

ContextHelper(List[Resource](), this.getClass.getName)

}

}

}

def search(target: ConceptNetwork): Option[SolvedIssue] = {

searcher.search(target, Nil)

}

/\*\*

\* Sets concepts to result to report.

\* @param identifier the result identifier in ShortTermMemory.

\* @param context ShortTermMemory to set understood Concepts to report.

\* @param issues found solved issues to set in ShortTermMemory.

\* @return updated ShortTermMemory

\*/

def setResultsToReport(identifier: String, context: ShortTermMemory, issues: List[SolvedIssue]): ShortTermMemory = {

val foundSolutions = Narrative[SolvedIssue](identifier, issues)

context.solutionsToReport = context.solutionsToReport + foundSolutions

context

}

}

package tu.coreservice.action.way2think

import cry4help.Cry4HelpWay2Think

import tu.model.knowledge.annotator.AnnotatedPhrase

import tu.model.knowledge.domain.{ConceptLink, Concept, ConceptNetwork}

import tu.model.knowledge.communication.{ContextHelper, ShortTermMemory}

import tu.model.knowledge.Resource

import tu.model.knowledge.primitive.KnowledgeString

/\*\*

\* @author max talanov

\* date 2012-07-03

\* time: 12:01 PM

\*/

trait SimulationReformulationAbstract {

/\*\*

\* Filters list of concepts, returns only concepts found in model.

\* @param currentConcepts List[Concept] to filter.

\* @param model ConceptNetwork to be filtered at.

\* @return List[Concept] filtered model concepts.

\*/

def filterConceptList(currentConcepts: List[Concept], model: ConceptNetwork): List[Concept] = {

val concepts: List[Concept] = currentConcepts.filter {

c => {

model.getNodeByName(c.uri.name).size > 0

}

}

concepts

}

/\*\*

\* Filters list of concepts, returns only concepts not found in model.

\* @param currentConcepts List[Concept] to filter.

\* @param model ConceptNetwork to be filtered at.

\* @return List[Concept] filtered.

\*/

def filterConceptListNegative(currentConcepts: List[Concept], model: ConceptNetwork): List[Concept] = {

val concepts: List[Concept] = currentConcepts.filter {

c => {

model.getNodeByName(c.uri.name).size < 1

}

}

concepts

}

/\*\*

\* Really stupid method to process ambiguity.

\* @param in List[Concept]

\* @return ShortTermMemory of Cry4HelpWay2Think

\* @deprecated

\*/

def processAmbiguous(in: List[AnnotatedPhrase]): ShortTermMemory = {

// ambiguity

var res: List[Resource] = in

res = res ++ List[Resource](KnowledgeString("Please clarify ambiquity", "Please.clarify.ambiquity"))

val context = ContextHelper.apply(res, "ambiguity")

val cry4helpWay2Think = new Cry4HelpWay2Think()

cry4helpWay2Think.apply(context)

}

/\*\*

\* Counts links of the specified concept to concepts of the List

\* @param concept to count links.

\* @param list List[Concept] to filter links of the specified concept.

\* @return Int number of filtered links

\*/

def countLinks(concept: Concept, list: List[Concept]): Int = {

concept.links.filter {

link: ConceptLink => {

if (link.source == concept) {

val destination = link.destination

val references: List[Concept] = list.filter {

c: Concept => c equals destination

}

references.size > 0

} else {

val source = link.source

val references: List[Concept] = list.filter {

c: Concept => c equals source

}

references.size > 0

}

}

}.size

}

/\*\*

\* Creates ConceptNetwork using Concept instances crated from concepts parameter and their links.

\* @param concepts List[Concept] to process.

\* @param name String name of ConceptNetwork.

\* @return ConceptNetwork.

\*/

def instantiateConcepts(concepts: List[Concept], name: String, simulationModel: ConceptNetwork): ConceptNetwork = {

var processedConcepts: List[Concept] = List[Concept]()

val instancesLinks: List[Pair[Concept, List[ConceptLink]]] = concepts.map(

(concept: Concept) => {

val currentInstance = Concept.createInstanceConcept(concept)

val incidentLinksNotProcessed: List[ConceptLink] = concept.links.filter {

link: ConceptLink => {

if (simulationModel.getLinkByName(link.uri.name).size > 0) {

if (link.source == concept) {

!processedConcepts.contains(link.destination)

} else if (link.destination == concept) {

!processedConcepts.contains(link.source)

} else {

false

}

} else {

false

}

}

}

val modelConcepts = simulationModel.getNodeByName(concept.uri.name)

val modelLinksNotProcessed: List[ConceptLink] = if (modelConcepts.size > 0) {

val mLinks: List[ConceptLink] = modelConcepts.head.links.filter(

(link: ConceptLink) => {

!incidentLinksNotProcessed.contains(link)

}

)

val gLinks: List[ConceptLink] = getGeneralisedLinks(modelConcepts.head, concepts)

val gFLinks: List[ConceptLink] = if (gLinks.size > 0) {

gLinks.filter {

g: ConceptLink => !incidentLinksNotProcessed.contains(g)

}

} else {

List[ConceptLink]()

}

mLinks ::: gFLinks

} else {

List[ConceptLink]()

}

val notProcessedLinks = incidentLinksNotProcessed ::: modelLinksNotProcessed

val linkInstances = notProcessedLinks.map {

link: ConceptLink => {

if (link.source == concept) {

val currentDestination = Concept.createInstanceConcept(link.destination)

ConceptLink.createInstanceConceptLink(link, currentInstance, currentDestination)

} else {

val currentSource = Concept.createInstanceConcept(link.source)

ConceptLink.createInstanceConceptLink(link, currentSource, currentInstance)

}

}

}

currentInstance.links = linkInstances

processedConcepts = processedConcepts ++ List(concept)

(currentInstance, linkInstances)

}

)

val instances: List[Concept] = instancesLinks.map {

i: Pair[Concept, List[ConceptLink]] => {

i.\_1

}

}

val links: List[List[ConceptLink]] = instancesLinks.map {

i: Pair[Concept, List[ConceptLink]] => {

i.\_2

}

}

val flatLinks: List[ConceptLink] = links.flatten

ConceptNetwork(instances, flatLinks, name)

}

def getGeneralisedLinks(concept: Concept, incidentConcepts: List[Concept]): List[ConceptLink] = {

val gLinks: List[ConceptLink] = concept.generalisations.frames.values.map {

c: Concept =>

this.getGeneralisedLinks(c, incidentConcepts)

}.toList.flatten

val filteredConceptLinks = concept.links.filter {

l: ConceptLink => {

if (l.source == concept) {

selectWithGeneralisations(l.destination, incidentConcepts).size > 0

} else {

selectWithGeneralisations(l.source, incidentConcepts).size > 0

}

}

}

filteredConceptLinks ::: gLinks

}

def selectWithGeneralisations(concept: Concept, checkList: List[Concept]): List[Concept] = {

checkList.filter {

c: Concept => {

c.getGeneralisationsRec.filter {

cG: Concept => cG.uri.name == concept.uri.name

}.size > 0 || c.uri.name == concept.uri.name

}

}

}

}

package tu.coreservice.action.way2think

import tu.model.knowledge.{Constant, SolvedIssue}

import tu.model.knowledge.domain.{ConceptLink, Concept, ConceptNetwork}

import tu.dataservice.knowledgebaseserver.KBAdapter

import org.slf4j.LoggerFactory

/\*\*

\* @author adel chepkunov, max talanov

\* Date: 10.07.12

\* Time: 7:21

\*/

//class Solutions(\_uri: KnowledgeURI, \_probability: Probability) extends Resource(\_uri, \_probability) {

//def this() = { this(new KnowledgeURI("defaultNamespace", "Solutions", "rev"), new Probability() ) }

//}

class Solutions {

val log = LoggerFactory.getLogger(this.getClass)

var solutions: List[SolvedIssue] = Nil

def add(item: SolvedIssue): List[SolvedIssue] = {

KBAdapter.solutionsAdd(item)

solutions = item :: solutions

solutions

}

/\*\*

\* Runs search of the solution based on relevance of issue and list of found and rejected solutions.

\* @param issue the ConceptNetwork of the issue.

\* @param badSolutions list of rejected solutions.

\* @return Option[SolvedIssue] if solution is found.

\*/

def search(issue: ConceptNetwork, badSolutions: List[ConceptNetwork]): Option[SolvedIssue] = {

val res = searchNotEffective(issue, badSolutions, 1)

if (res.size == 0) {

None

} else {

Some(res.sortWith((s, t) => s.issue.nodes.size < t.issue.nodes.size).head)

}

}

/\*\*

\* Not effective search based on issue ConceptNetwork and list of rejected Solutions.

\* @param issue ConceptNetwork describing issue

\* @param badSolutions found and rejected Solutions

\* @param errors number of errors allowed for solution.

\* @return List[SolvedIssue] of found solutions.

\*/

private def searchNotEffective(issue: ConceptNetwork, badSolutions: List[ConceptNetwork], errors: Int): List[SolvedIssue] = {

var found\_solutions\_map: Map[Double,SolvedIssue] = Map()

var found\_solutions: List[SolvedIssue] = List()

for (s <- solutions) {

found\_solutions\_map+=((relevance(issue, s.issue),s));

/\*if (relevance(issue, s.issue) <= Constant.DistanceThreadHold) {

found\_solutions = s :: found\_solutions

}\*/

}

found\_solutions=List(found\_solutions\_map.minBy(\_.\_1).\_2);

//If not found, then return None else - return SolvedIssue with minimal size

val res: List[SolvedIssue] = if (errors == 0 || !found\_solutions.isEmpty) {

found\_solutions

} else {

val subIssues = CNMinusList(issue)

subIssues.map(p => searchNotEffective(p, badSolutions, errors - 1)).reduce((A, B) => A ::: B)

}

log info("solutions found={}", res)

res

}

/\*\*

\* Creates list of ConceptNetworks reduced by one concept of specified issue ConceptNetwork.

\* @param issue ConceptNetwork to create reduced ConceptNetworks List.

\* @return reduced ConceptNetworks List.

\*/

private def CNMinusList(issue: ConceptNetwork): List[ConceptNetwork] = {

def issueWithoutNode(n: Concept): ConceptNetwork = {

val nodes = issue.rootNodes.filter(p => p != n)

new ConceptNetwork(nodes, issue.links, issue.uri, issue.probability)

}

issue.rootNodes.map(issueWithoutNode)

}

//TODO: allow K errors

/\*\*

\* Calculates relevance of the specified issue and master ConceptNetwork-s.

\* @param issue ConceptNetwork to calculate distance.

\* @param master ConceptNetwork to compare with.

\* @param errors Int number of errors to be ok, currently not used.

\* @return Int distance of specified issue to master ConceptNetwork-s.

\*/

def distance(issue: ConceptNetwork, master: ConceptNetwork, errors: Int): Int = {

//find first node

val listWithFirstNode = master.rootNodes.filter(p => compareWithGeneralisation(p, issue.rootNodes(0)))

if (listWithFirstNode.isEmpty) {

return 1

}

val depth = issue.rootNodes.size

if (!masterHasAllLinks(issue.rootNodes(0), issue, listWithFirstNode(0), master, depth)) {

return 1

}

0

}

/\*\*

\* Calculates relevance of specified issue and master ConceptNetworks, each missing issue Concept add 0.1 each master missing concept adds 1.

\* @param issue ConceptNetwork to compare with master.

\* @param master ConceptNetwork to be compared with.

\* @return Double relevance calculated according to missing concepts of issue and master.

\*/

def relevance(issue: ConceptNetwork, master: ConceptNetwork): Double = {

val missingLists = findMissing(issue, master)

val sum: Double = missingLists.\_1.size \* Constant.IssueMissingFactor + missingLists.\_2.size \* Constant.DomainMissingFactor

sum

}

/\*\*

\* Searches for missing Concept-s of issue ConceptNetwork and master ConceptNetwork, comparing list of concepts according to generalisation of issue Concept-s to master Concept-s.

\* @param issue ConceptNetwork to search generalisations in master.

\* @param master ConceptNetwork to search in generalisations.

\* @return Pair of issue concepts missing and master concept missing .

\*/

def findMissing(issue: ConceptNetwork, master: ConceptNetwork): Pair[List[Concept], List[Concept]] = {

val issueNodes = issue.nodes

val masterNodes = master.nodes

val issueMissing = issueNodes.filter {

iC: Concept => {

masterNodes.filter {

mC: Concept => {

iC.hasExactGeneralisationRec(mC)

}

}.size <= 0

}

}

val masterMissing = masterNodes.filter {

mC: Concept => {

issueNodes.filter {

iC: Concept => {

iC.hasExactGeneralisationRec(mC)

}

}.size <= 0

}

}

(issueMissing, masterMissing)

}

/\*\*

\* Checks if masterConcept has same ConceptLinks as has all the links of specified issueConcept.

\* @param issueConcept to check with master.

\* @param issue

\* @param masterConcept

\* @param master

\* @param depth

\* @return

\*/

def masterHasAllLinks(issueConcept: Concept, issue: ConceptNetwork, masterConcept: Concept, master: ConceptNetwork, depth: Int): Boolean = {

if (depth == 0) return true

val links = getConceptChildren(issueConcept, issue.links)

if (links.isEmpty) return true

for (issueNext <- links) {

// forall for empty list always return true, so we use not (!masterHasAllLinks)

val bad = getConceptChildren(masterConcept, master.links).filter(p => compareWithGeneralisation(p, issueNext))

.forall(p => !masterHasAllLinks(issueNext, issue, p, master, depth - 1))

if (bad) return false

}

true

}

def getConceptChildren(concept: Concept, links: List[ConceptLink]): List[Concept] = {

links.filter(p => compareWithGeneralisation(concept, p.source)).map(p => p.destination) :::

links.filter(p => compareWithGeneralisation(concept, p.destination)).map(p => p.source)

}

def compareWithGeneralisation(p: Concept, q: Concept): Boolean = {

if (p.uri.name == q.uri.name) {

return true

}

for (pgkey <- p.uri :: p.\_generalisations.frames.keys.toList) {

for (qgkey <- q.uri :: q.\_generalisations.frames.keys.toList) {

if (pgkey.name == qgkey.name) {

return true

}

}

}

false

}

}

package tu.coreservice.action.way2think

import tu.model.knowledge.communication.ShortTermMemory

import tu.model.knowledge.{Constant, Probability, KnowledgeURI}

import tu.coreservice.action.{Action}

import tu.model.knowledge.helper.URIGenerator

import tu.model.knowledge.domain.Concept

import tu.model.knowledge.narrative.Narrative

/\*\*

\* Way2Think interface.

\* @author max Talanov, alexander toschev

\* date 2012-05-28

\* time: 11:09 PM

\*/

abstract class Way2Think(\_uri: KnowledgeURI, \_probability: Probability = new Probability())

extends Action(\_uri, \_probability) {

/\*\*

\* Way2Think interface.

\* @param inputContext ShortTermMemory of all inbound parameters.

\* @return outputContext

\*/

def apply(inputContext: ShortTermMemory): ShortTermMemory

def this() = this(URIGenerator.generateURI("Way2Think"))

/\*\*

\* Sets concepts to result to report.

\* @param context ShortTermMemory to set understood Concepts to report.

\* @param concepts understood concepts to set in ShortTermMemory.

\* @return updated ShortTermMemory

\*/

def setResultsToReport(context: ShortTermMemory, concepts: List[Concept]): ShortTermMemory = {

val understoodConcepts = Narrative[Concept](Constant.UNDERSTOOD\_CONCEPTS, concepts)

context.resultToReport = context.resultToReport + understoodConcepts

context

}

}

# Модуль coreservice.linkparser

package tu.coreservice.linkparser

import collection.JavaConversions.\_

import tu.coreservice.action.way2think.Way2Think

import tu.model.knowledge.communication.{ContextHelper, ShortTermMemory}

import tu.model.knowledge.{Constant, KnowledgeURI, Resource}

import tu.model.knowledge.annotator.{AnnotatedPhrase, AnnotatedSentence, AnnotatedNarrative}

import tu.coreservice.action.way2think.cry4help.Cry4HelpWay2Think

import relex.entity.EntityMaintainer

import relex.output.OpenCogScheme

import relex.ParsedSentence

import relex.feature.{FeatureNameFilter, FeatureNode}

import org.slf4j.LoggerFactory

import tu.model.knowledge.domain.{Concept, ConceptLink}

import tu.exception.{NoExpectedInformationException, UnexpectedException}

import tu.dataservice.knowledgebaseserver.Defaults

import tu.nlp.server.NLPFactory

import tu.exception.UnexpectedException

/\*\*

\* Processes AnnotatedSentence each AnnotatedSentence via RelationExtractorKB.

\* @author max talanov

\* Date: 7/31/12

\* Time: 4:18 AM

\*/

class LinkParser extends Way2Think {

val log = LoggerFactory.getLogger(this.getClass)

val relexServer = NLPFactory.createProcessor()

def start() = false

def stop() = false

/\*\*

\* Way2Think interface.

\* @param inputContext ShortTermMemory of all inbound parameters.

\* @return outputContext

\*/

def apply(inputContext: ShortTermMemory): ShortTermMemory = {

val narrative = getLastResult(inputContext)

val sentences: List[AnnotatedSentence] = narrative.sentences

val updatedSentences = processSentences(sentences, inputContext)

val updatedNarrative = AnnotatedNarrative(updatedSentences, KnowledgeURI(Constant.LINK\_PARSER\_RESULT\_NAME))

val outputContext = ContextHelper(List[Resource](updatedNarrative), updatedNarrative, this.getClass.getName)

outputContext

}

def getLastResult(inputContext: ShortTermMemory): AnnotatedNarrative = {

try {

inputContext.lastResult match {

case Some(narrative: AnnotatedNarrative) => narrative

case \_ => throw new NoExpectedInformationException("$Last\_result\_contains\_no\_narrative")

}

} catch {

case e: ClassCastException => {

val cry4Help = Cry4HelpWay2Think("$Context\_lastResult\_is\_not\_expectedType " + e.getMessage)

ContextHelper(List[Resource](cry4Help), cry4Help, this.getClass.getName + " result")

throw new NoExpectedInformationException("$Context\_lastResult\_is\_not\_expectedType " + e.getMessage)

}

}

}

/\*\*

\* Run through List of AnnotatedSentence and processes each AnnotatedSentence via RelationExtractorKB.

\* @param sentences to process.

\* @param context ShortTermMemory to store Errors in.

\* @return List of AnnotatedSentence.

\*/

def processSentences(sentences: List[AnnotatedSentence], context: ShortTermMemory): List[AnnotatedSentence] = {

sentences.map {

sentence: AnnotatedSentence => {

val parse: ParsedSentence = processSentenceRelex(sentence, sentences)

log debug("parse = {}", parse.toString)

val node: FeatureNode = new FeatureNode()

node.set("head", parse.getLeft.get("head"))

node.set("background", parse.getLeft.get("background"))

val res = processNode(node.get("head"), sentence)

res match {

case (None, Some(e: Error)) => {

context.errors = context.errors ::: List(e)

}

case (Some(c: Concept), None) => {

// no processing

}

case (Some(c: Concept), Some(e: Error)) => {

context.errors = context.errors ::: List(e)

}

case (None, None) => {

// no processing

}

}

}

}

sentences

}

/\*\*

\* Processes sentence with RelationExtractorKB that takes in account KBAnnotator results.

\* Run relex and extract sentences.

\* @param sentence to process via RelationExtractorKB

\* @param sentences processed

\* @return ProcessedSentence

\*/

def processSentenceRelex(sentence: AnnotatedSentence, sentences: List[AnnotatedSentence]): ParsedSentence = {

val relexSentence = relexServer.processSentence(sentence.text, sentences)

log debug("relexSentence ={}", relexSentence)

val parsesNum = relexSentence.getNumParses

if (parsesNum < 1) {

throw new UnexpectedException("$No\_parses\_produced")

} else {

relexSentence.getParses.get(0)

}

}

/\*\*

\* Gets Concept from sentence if not found, creates orphan Concept and returns Error, if found returns Concept with no Error, otherwise only Error is returned.

\* @param feature FeatureNode to process: find AnnotatedPhrase -> Concept,

\* @param sentence to find FeatureNode in.

\* @return Pair of Concept and Error

\*/

def processNode(feature: FeatureNode, sentence: AnnotatedSentence): Pair[Option[Concept], Option[Error]] = {

log trace("processNode(feature={})", feature)

try {

val nameOrigString: Pair[Option[String], Option[String]] = getNameOrigString(feature)

val name: String = nameOrigString match {

case Pair(Some(n: String), \_) => n

case Pair(None, Some(oS: String)) => oS

case Pair(None, None) => throw new UnexpectedException("$No\_name\_found")

}

val phraseConceptError: Triple[AnnotatedPhrase, Option[Concept], Option[Error]] = getConcept(name, sentence)

phraseConceptError match {

case Triple(annotatedPhrase: AnnotatedPhrase, Some(concept: Concept), None) => {

val updatedConcept = addLinks(feature, concept, annotatedPhrase, sentence)

log debug("updated concept={}, it was={}", updatedConcept, concept)

(Some(updatedConcept), None)

}

case Triple(annotatedPhrase: AnnotatedPhrase, Some(concept: Concept), Some(e: Error)) => {

val updatedConcept = addLinks(feature, concept, annotatedPhrase, sentence)

log debug("updated concept={}, it was={}", updatedConcept.toString, concept.toString)

(Some(updatedConcept), None)

}

case Triple(annotatedPhrase: AnnotatedPhrase, None, Some(e: Error)) => {

log debug("produced parsing error={}", e)

(None, Some(e))

}

}

} catch {

case e: RuntimeException => {

log error e.getMessage

throw new UnexpectedException("$Wrong\_feature\_requested " + e.getMessage)

}

}

}

/\*\*

\* Updates ConceptLink-s of tense and pos adding links.

\* @param feature based on it the Concept's ConceptLink-s are updated.

\* @param concept Concept to update ConceptLink-s

\* @param annotatedPhrase AnnotatedPhrase to add POS and tense.

\* @param sentence to recursively process FeatureNode-s and ConceptLink-s.

\* @return updated Concept.

\*/

private def addLinks(feature: FeatureNode, concept: Concept, annotatedPhrase: AnnotatedPhrase, sentence: AnnotatedSentence): Concept = {

if (feature.get("tense") != null) {

log trace "tense=" + feature.get("tense").getValue

annotatedPhrase.tense = feature.get("tense").getValue

log debug("added tense={} to phrase={}", feature.get("tense").getValue, annotatedPhrase)

}

if (feature.get("pos") != null) {

log trace "pos=" + feature.get("pos").getValue

annotatedPhrase.pos = feature.get("pos").getValue

log debug("added pos={} to phrase={}", feature.get("pos").getValue, annotatedPhrase)

}

if (feature.get("links") != null) {

// log debug "links=" + feature.get("links")

log trace "==>"

val processedLinks = processLink(feature.get("links"), concept, sentence)

concept.links = concept.links ::: processedLinks

}

val next = feature.get("NEXT")

if (next != null) {

log trace "=>"

processNode(next, sentence)

}

concept

}

/\*\*

\* Searches for FeatureNode with specified name.

\* @param feature FeatureNode to start search.

\* @param nameToFind the String name to find.

\* @return first found FeatureNode or None.

\*/

private def findFeatureNode(feature: FeatureNode, nameToFind: String): Option[FeatureNode] = {

val name: String = getName(feature)

if (name == nameToFind) {

Some(feature)

} else {

if (feature.get("links") != null) {

log trace "==>"

val processedLinks = findFeatureInLinks(feature.get("links"), nameToFind)

processedLinks

}

val next = feature.get("NEXT")

if (next != null) {

log trace "=>"

findFeatureNode(next, nameToFind)

}

None

}

}

private def findFeatureInLinks(feature: FeatureNode, nameToFind: String): Option[FeatureNode] = {

val filteredFeatures = feature.getFeatureNames.filter {

name: String => Constant.RelexFeatures.contains(name)

}

if (filteredFeatures.size > 0) {

val foundNodeString: Option[String] = filteredFeatures.toList.find {

(name: String) => {

val found = findFeatureNode(feature.get(name), nameToFind)

found match {

case Some(f) => {

true

}

case Pair(None, Some(e: Error)) => {

false

}

}

}

}

foundNodeString match {

case Some(featureName) => {

val found: Option[FeatureNode] = findFeatureNode(feature.get(featureName), nameToFind)

found

}

case None => None

}

} else {

None

}

}

/\*\*

\* Searches for AnnotatedPhrase via specified name in specified AnnotatedSentence, if no Concept found new orphan Concept is created and Error is set, if more than 1 Concept found Error returned.

\* @param name to be used to search for a AnnotatedPhrase.

\* @param sentence to search AnnotatedPhrase in.

\* @return Concept and Error pair.

\*/

def getConcept(name: String, sentence: AnnotatedSentence): Triple[AnnotatedPhrase, Option[Concept], Option[Error]] = {

log trace("getConcept(name = {})", name)

val phrases = findPhrase(name, sentence)

if (phrases.size == 1) {

val phrase = phrases.head

val concepts = phrase.concepts

if (concepts.size == 1) {

val concept = concepts.head

log trace("returned concept={}", concept)

(phrase, Some(concept), None)

} else if (concepts.size < 1) {

val concept = Concept.createInstanceConcept(phrase)

phrase.conceptsAdd(concept)

concept.phrasesAdd(phrase)

log trace("created new concept={}", concept)

log trace("added it to phrase={}", phrase)

(phrase, Some(concept), Some(new Error("$No\_concepts\_found\_for\_phrase: " + phrase)))

} else {

log info ("concepts were ambiguous")

(phrase, None, Some(new Error("$Ambiguous\_concepts")))

}

} else if (phrases.size < 1) {

throw new UnexpectedException("$No\_phrases\_found " + name)

} else {

throw new UnexpectedException("$Ambiguous\_phrases " + name)

}

}

def getName(feature: FeatureNode): String = {

if (feature.get("name") != null) {

val name = feature.get("name").getValue

log trace "name=" + feature.get("name").getValue

name

} else if (feature.get("orig\_str") != null) {

val origStr = feature.get("orig\_str").getValue

log trace "orig\_str" + feature.get("orig\_str").getValue

origStr

} else {

throw new UnexpectedException("$No\_name\_specified")

}

}

/\*\*

\* Returns Pair of name and origString if found in specified FeatureNode.

\* @param feature FeatureNode to process

\* @return Pair of Option name and Option origString

\*/

def getNameOrigString(feature: FeatureNode): Pair[Option[String], Option[String]] = {

val optionName: Option[String] = if (feature.get("name") != null) {

val name = feature.get("name").getValue

log trace "name=" + feature.get("name").getValue

Some(name)

} else {

None

}

val optionOrigString: Option[String] = if (feature.get("orig\_str") != null) {

val origStr = feature.get("orig\_str").getValue

log trace "orig\_str" + feature.get("orig\_str").getValue

Some(origStr)

} else if (feature.get("nameSource") != null && feature.get("nameSource").get("orig\_str") != null) {

val origStr = feature.get("nameSource").get("orig\_str").getValue

log trace "orig\_str" + feature.get("nameSource").get("orig\_str").getValue

Some(origStr)

} else {

None

}

(optionName, optionOrigString)

}

def findPhrase(value: String, sentence: AnnotatedSentence): List[AnnotatedPhrase] = {

val underscoreLess: String = value.replaceAll("\_", " ")

val filteredPhrase: List[AnnotatedPhrase] = sentence.phrases.filter {

phrase: AnnotatedPhrase => {

phrase.findPhrase(underscoreLess.trim.toLowerCase) match {

case Some(ph: AnnotatedPhrase) => true

case None => false

}

}

}

filteredPhrase

}

def processLink(feature: FeatureNode, source: Concept, sentence: AnnotatedSentence): List[ConceptLink] = {

log trace("processLink(feature: {}", feature.get("name"))

log trace("feature names={}", feature.getFeatureNames)

try {

val filteredFeatures = feature.getFeatureNames.filter {

name: String => Constant.RelexFeatures.contains(name)

}

log trace("filteredFeatures ={}", filteredFeatures)

if (filteredFeatures.size > 0) {

val filteredDestinationFeatures = filteredFeatures.toList.filter {

(name: String) => {

val destinationError = processNode(feature.get(name), sentence)

destinationError match {

case Pair(Some(destination), None) => {

true

}

case Pair(None, Some(e: Error)) => {

false

}

}

}

}

val conceptLinks: List[ConceptLink] = filteredDestinationFeatures.toList.map(

(name: String) => {

val destinationError = processNode(feature.get(name), sentence)

destinationError match {

case Pair(Some(destination), None) => {

val conceptLinkName = if (name.indexOf("\_") > -1) {

name.substring(1)

} else {

name

}

ConceptLink(source, destination, conceptLinkName)

}

}

}

)

conceptLinks

} else {

log.warn("$No\_links\_found")

//throw new UnexpectedException("$No\_links\_found")

List[ConceptLink]()

}

} catch {

case e: RuntimeException => {

throw new UnexpectedException("$No\_features\_found " + e.getMessage)

}

}

}

def linksFilter: FeatureNameFilter = {

val ignores: java.util.HashSet[String] = new java.util.HashSet[String]

ignores.add("nameSource")

ignores.add("syn\_location")

ignores.add("SIG")

ignores.add("linkR0")

ignores.add("linkR1")

ignores.add("linkR2")

ignores.add("linkR3")

ignores.add("linkR4")

ignores.add("linkR5")

ignores.add("linkR6")

ignores.add("linkR7")

ignores.add("linkL0")

ignores.add("linkL1")

ignores.add("linkL2")

ignores.add("linkL3")

ignores.add("linkL4")

ignores.add("linkL5")

ignores.add("linkL6")

ignores.add("linkL7")

ignores.add("first\_verb")

ignores.add("HEAD-FLAG")

ignores.add("POS")

ignores.add("head-word")

ignores.add("morph")

ignores.add("num")

ignores.add("num\_left\_links")

ignores.add("num\_right\_links")

ignores.add("str")

ignores.add("ref")

ignores.add("subj")

ignores.add("obj")

ignores.add("iobj")

ignores.add("this")

ignores.add("wall")

ignores.add("COMP-FLAG")

ignores.add("VTAlg\_flag")

ignores.add("comparative-name")

ignores.add("comparative-nameSource")

ignores.add("comparative-obj")

ignores.add("comparative-obj-name")

ignores.add("comparative-obj-nameSource")

ignores.add("comp-obj-copy")

ignores.add("name")

ignores.add("tense")

ignores.add("PREP-FLAG")

ignores.add("links")

ignores.add("")

ignores.add("head")

ignores.add("background")

ignores.add("words")

val order: java.util.ArrayList[String] = new java.util.ArrayList[String]

Constant.RelexFeatures.map {

name: String => order.add(name)

}

new FeatureNameFilter(ignores, order)

}

}

package tu.coreservice.linkparser

import relex.morphy.{MorphyJWNL, Morphed}

import tu.model.knowledge.communication.ShortTermMemory

import tu.model.knowledge.annotator.{AnnotatedPhrase, AnnotatedSentence}

import java.rmi.UnexpectedException

import tu.model.knowledge.frame.Frame

import tu.model.knowledge.{Constant, Resource, KnowledgeURI}

import tu.model.knowledge.domain.Concept

import org.slf4j.LoggerFactory

/\*\*

\* @author alex toschev, max talanov

\* Time stamp: 7/25/12 5:40 PM

\*/

class MorphyKB(var \_sentences: List[AnnotatedSentence]) extends MorphyJWNL {

val log = LoggerFactory.getLogger(this.getClass)

def this() {

this(List[AnnotatedSentence]())

}

def sentences = \_sentences

def sentences\_=(in: List[AnnotatedSentence]) {

\_sentences = in

}

/\*Error case

[22.11 23:29:21] [main] INFO MorphyKB - res =Browser N:Browser

[22.11 23:29:21] [main] INFO MorphyKB - res =is V:be

[22.11 23:29:21] [main] INFO MorphyKB - res =an N:an

[22.11 23:29:21] [main] INFO MorphyKB - res =object N:object V:object

[22.11 23:29:21] [main] INFO MorphyKB - res =.

\*/

/\* Correct case

[22.11 23:35:47] [main] INFO MorphyKB - res =Browser N:Browser

[22.11 23:35:47] [main] INFO MorphyKB - res =is V:be

[22.11 23:35:47] [main] INFO MorphyKB - res =object N:object V:object

[22.11 23:35:47] [main] INFO MorphyKB - res =.

[22.11 23:35:47] [main] INFO MorphyKB - res =an N:an

\*/

override def morph(word: String): Morphed = {

val res: Morphed = super.morph(word)

val foundPhrases: List[AnnotatedPhrase] = searchWord(word, \_sentences)

if (foundPhrases.size < 1) {

log debug ("res ={}", res)

res

} else if (foundPhrases.size == 1) {

//unambiguous case

// get concepts and correct annotation

val mostGenericConcepts: List[Concept] = foundPhrases(0).concepts.map {

concept: Concept => {

mostGenericGeneralisation(concept)

}

}.flatten

/\* if (cat.equals("noun"))

m.putRoot(NOUN\_F, root);

else if (cat.equals("verb")) {

if (negativeVerb)

m.putRootNegative(VERB\_F, root);

else

m.putRoot(VERB\_F, root);

} else if (cat.equals("adj"))

m.putRoot(ADJ\_F, root);

else if (cat.equals("adv"))

m.putRoot(ADV\_F, root);

else

throw new RuntimeException("Unknown WordNet category: [" + cat + "] with root [" + root + "]"); \*/

if (!getConceptByName(mostGenericConcepts, Constant.subjectConceptName).isEmpty) {

res.getFeatures.clear()

res.putRoot(Constant.NOUN\_F, word)

} else if (!getConceptByName(mostGenericConcepts, Constant.objectConceptName).isEmpty) {

res.getFeatures.clear()

res.putRoot(Constant.NOUN\_F, word)

} else if (!getConceptByName(mostGenericConcepts, Constant.actionConceptName).isEmpty) {

res.getFeatures.clear()

res.putRoot(Constant.VERB\_F, word)

} else if (!getConceptByName(mostGenericConcepts, Constant.desireConceptName).isEmpty) {

res.getFeatures.clear()

res.putRoot(Constant.VERB\_F, word)

} else if (!getConceptByName(mostGenericConcepts, Constant.formOfPoliteness).isEmpty) {

res.getFeatures.clear()

res.putRoot(Constant.ADV\_F, word)

}

res

} else {

throw new UnexpectedException("$Ambiguous\_case")

}

}

/\*\*

\* Gets source by name from List of Concept-s

\* @param concepts to find Concept.

\* @param name to filter concepts.

\* @return Option[Concept].

\*/

def getConceptByName(concepts: List[Concept], name: String): Option[Concept] = {

concepts.find {

concept: Concept => {

concept.uri.name == (name + "Concept")

}

}

}

/\*\*

\* Get most generic generalisation of specified source, if no generalisations found in current source, current source is returned.

\* @param concept to process.

\* @return most generic source.

\*/

def mostGenericGeneralisation(concept: Concept): List[Concept] = {

if (concept.generalisations.frames.size < 1) {

List(concept)

} else {

mostGenericGeneralisation(concept.generalisations.frames.values.toList)

}

}

def mostGenericGeneralisation(concepts: List[Concept]): List[Concept] = {

val nonGeneralisedConcepts = concepts.filter {

concept: Concept => {

concept.generalisations.frames.size == 0

}

}

val generalisedConcepts = concepts.filter {

concept: Concept => {

concept.generalisations.frames.size > 0

}

}

val mostGenericConcepts = generalisedConcepts.map {

concept: Concept => {

mostGenericGeneralisation(concept.generalisations.frames.values.toList)

}

}.flatten

mostGenericConcepts ::: nonGeneralisedConcepts

}

def searchWord(aWord: String, sentences: List[AnnotatedSentence]): List[AnnotatedPhrase] = {

val phrases: List[AnnotatedPhrase] = sentences.map {

sentence: AnnotatedSentence => {

sentence.phrases

}

}.flatten

phrases.filter {

phrase: AnnotatedPhrase => {

phrase.phrases.count {

word: AnnotatedPhrase => {

word.text == aWord

}

} > 0 || phrase.text == aWord

}

}

}

def processLastResult(context: ShortTermMemory): List[AnnotatedSentence] = {

val lastResult = context.lastResult

lastResult match {

case Some(frame: Frame) => {

val resourcesSentences: List[AnnotatedSentence] = frame.resources.filter {

uriResource: Pair[KnowledgeURI, Resource] => {

uriResource.\_2 match {

case aS: AnnotatedSentence => {

true

}

case \_ => false

}

}

}.map {

uriResource: Pair[KnowledgeURI, Resource] => {

uriResource.\_2.asInstanceOf[AnnotatedSentence]

}

}.toList

resourcesSentences

}

case \_ => {

throw new UnexpectedException("$Last\_result\_contains\_unexpected\_type " + lastResult.getClass.getName)

}

}

}

}

package tu.coreservice.linkparser

import scala.collection.JavaConversions.\_

import java.util.ArrayList

import java.util.TreeMap

import relex.algs.SentenceAlgorithmApplier

import relex.anaphora.Antecedents

import relex.anaphora.Hobbs

import relex.chunk.ChunkRanker

import relex.chunk.LexChunk

import relex.concurrent.RelexContext

import relex.entity.EntityMaintainer

import relex.entity.EntityTagger

import relex.morphy.Morphy

import relex.morphy.MorphyFactory

import relex.parser.LGParser

import relex.parser.LocalLGParser

import relex.parser.RemoteLGParser

import relex.stats.TruthValue

import relex.stats.SimpleTruthValue

import relex.tree.{PhraseTree, PhraseMarkup}

import relex.{Sentence, Document, ParseStats, Version}

import org.slf4j.LoggerFactory

import relex.feature.FeatureNode

import tu.model.knowledge.annotator.AnnotatedSentence

/\*\*

\* Based on RelationExtractor

\* @author max talanov

\* date 2012-07-28

\* time: 5:21 PM

\*/

class RelationExtractorKB(useSocket: Boolean, sentences: List[AnnotatedSentence]) {

val log = LoggerFactory.getLogger(this.getClass)

/\*\*Syntax processing \*/

private val parser: LGParser = new LocalLGParser /\* (useSocket) {

new RemoteLGParser()

} else {

new LocalLGParser()

} \*/

parser.getConfig.setStoreConstituentString(true)

parser.getConfig.setLoadSense(true)

/\*\*The LinkParserClient to be used - this class isn't thread safe! \*/

val morphy: MorphyKB = MorphyFactory.getImplementation("tu.coreservice.linkparser.MorphyKB").asInstanceOf[MorphyKB]

// val morphy: Morphy = MorphyFactory.getImplementation(MorphyFactory.DEFAULT\_SINGLE\_THREAD\_IMPLEMENTATION)

// morphy.sentences = sentences

private var context: RelexContext = new RelexContext(parser, morphy)

/\*\*Dependency processing \*/

private var sentenceAlgorithmApplier: SentenceAlgorithmApplier = new SentenceAlgorithmApplier()

/\*\*Penn tree-bank style phrase structure markup. \*/

private var phraseMarkup: PhraseMarkup = new PhraseMarkup()

var do\_tree\_markup: Boolean = false

/\*\*Anaphora resolution \*/

var antecedents: Antecedents = new Antecedents()

private var hobbs: Hobbs = new Hobbs(antecedents)

var do\_anaphora\_resolution: Boolean = false

/\*\*Document - holder of sentences \*/

private var doco: Document = new Document()

/\*\*Stanford parser compatibility mode \*/

var do\_stanford: Boolean = false

/\*\*Penn tagset compatibility mode \*/

var do\_penn\_tagging: Boolean = false

/\*\*Expand preposition markup to two dependencies. \*/

var do\_expand\_preps: Boolean = false

/\*\*Perform entity substitution before parsing \*/

var do\_pre\_entity\_tagging: Boolean = false

/\*\*Perform entity tagging after parse \*/

var do\_post\_entity\_tagging: Boolean = false

var tagger: EntityTagger = null

/\*\*Statistics \*/

private var stats: ParseStats = new ParseStats()

var \_starttime: Long = System.currentTimeMillis

private var sumtime: TreeMap[String, Long] = new TreeMap[String, Long]()

private var cnttime: TreeMap[String, Long] = new TreeMap[String, Long]()

def startime = \_starttime

//init(useSocket)

def this() {

this(false, List[AnnotatedSentence]())

}

private def prt\_chunks(chunks: List[Nothing]) {

for (ch <- chunks) {

log.debug(ch.toString)

}

log.debug("\n======\n")

}

private def discriminate(ranker: ChunkRanker) {

val chunks: ArrayList[LexChunk] = ranker.getChunks

for (ch <- chunks) {

val sz: Int = ch.size

var weight: Double = sz - 3

if (weight < 0) weight = -weight

weight = 1.0 - 0.2 \* weight

val tv: TruthValue = ch.getTruthValue

val stv: SimpleTruthValue = tv.asInstanceOf[SimpleTruthValue]

var confidence: Double = stv.getConfidence

confidence \*= weight

stv.setConfidence(confidence)

}

}

final val verbosity: Int = 1

final val DEFAULT\_MAX\_PARSES: Int = 100

final val DEFAULT\_MAX\_SENTENCE\_LENGTH: Int = 1024

final val DEFAULT\_MAX\_PARSE\_SECONDS: Int = 30

final val DEFAULT\_MAX\_PARSE\_COST: Int = 1000

private def init(useSocket: Boolean) {

parser.getConfig.setStoreConstituentString(true)

parser.getConfig.setLoadSense(true)

//val morphy: MorphyKB = MorphyFactory.getImplementation("tu.coreservice.linkparser.MorphyKB").asInstanceOf[MorphyKB]

// val morphy: Morphy = MorphyFactory.getImplementation(MorphyFactory.DEFAULT\_SINGLE\_THREAD\_IMPLEMENTATION)

//morphy.sentences = sentences

context = new RelexContext(parser, morphy)

sentenceAlgorithmApplier = new SentenceAlgorithmApplier()

setMaxParses(DEFAULT\_MAX\_PARSES)

setMaxParseSeconds(DEFAULT\_MAX\_PARSE\_SECONDS)

setMaxCost(DEFAULT\_MAX\_PARSE\_COST)

phraseMarkup = new PhraseMarkup()

antecedents = new Antecedents()

hobbs = new Hobbs(antecedents)

do\_anaphora\_resolution = false

doco = new Document()

do\_tree\_markup = false

do\_stanford = true

do\_penn\_tagging = false

do\_expand\_preps = false

do\_pre\_entity\_tagging = false

do\_post\_entity\_tagging = false

tagger = null

stats = new ParseStats()

sumtime = new TreeMap[String, Long]()

cnttime = new TreeMap[String, Long]()

}

def getVersion: String = {

parser.getVersion + "\t" + Version.getVersion

}

/\*\*

\* Set the max number of parses.

\* This will NOT reduce processing time; all parses are still computed,

\* but only this many are returned.

\*/

def setMaxParses(maxParses: Int) {

parser.getConfig.setMaxLinkages(maxParses)

}

def setMaxCost(maxCost: Int) {

parser.getConfig.setMaxCost(maxCost)

}

def setAllowSkippedWords(allow: Boolean) {

parser.getConfig.setAllowSkippedWords(allow)

}

def setMaxParseSeconds(maxParseSeconds: Int) {

parser.getConfig.setMaxParseSeconds(maxParseSeconds)

}

/\*\*

\* Clear out the cache of old sentences.

\*

\* The Anaphora resolver keeps a list of sentences previously seen,

\* so that anaphora resolution can be done. When starting the parse

\* of a new text, this cache needs to be cleaned out. This is the

\* way to do so.

\*/

def clear() {

antecedents.clear()

hobbs = new Hobbs(antecedents)

}

def processSentence(sentence: String): Sentence = {

processSentence(sentence, null)

}

/\*\*

\* Process one sentence with entityMaintainer already setup.

\* @param sentence to process via EntityMaintainer.

\* @param \_entityMaintainer EntityMaintainer

\* @return processed Sentence

\*/

def processSentence(sentence: String, \_entityMaintainer: EntityMaintainer): Sentence = {

//todo repeatable work as LinkParserTest

\_starttime = System.currentTimeMillis

var entityMaintainer = \_entityMaintainer

if (entityMaintainer == null) {

entityMaintainer = new EntityMaintainer()

}

var sntc: Sentence = null

try {

if (verbosity > 0) {

\_starttime = System.currentTimeMillis

}

sntc = parseSentence(sentence, entityMaintainer)

if (verbosity > 0) {

reportTime("Link-parsing: ")

}

for (parse <- sntc.getParses) {

if (do\_expand\_preps) {

parse.getLeft.set("expand-preps", new FeatureNode("T"))

}

if (do\_post\_entity\_tagging && (tagger != null)) {

tagger.tagEntities(sentence)

tagger.tagParse(parse)

}

entityMaintainer.tagConvertedSentence(parse)

sentenceAlgorithmApplier.applyAlgs(parse, context)

if (do\_stanford) sentenceAlgorithmApplier.extractStanford(parse, context)

if (do\_penn\_tagging) sentenceAlgorithmApplier.pennTag(parse, context)

entityMaintainer.repairSentence(parse.getLeft)

if (do\_tree\_markup) {

phraseMarkup.markup(parse)

val pt: PhraseTree = new PhraseTree(parse.getLeft)

parse.setPhraseString(pt.toString)

}

}

sntc.simpleParseRank()

if (do\_anaphora\_resolution) {

hobbs.addParse(sntc)

hobbs.resolve(sntc)

}

}

catch {

case e: Exception => {

log.error("Error: Failed to process sentence: " + sentence)

e.printStackTrace()

}

}

if (verbosity > 0) reportTime("RelEx processing: ")

sntc

}

/\*\*

\* Parses a sentence, using the parser. The private ArrayList of

\* currentParses is filled with the ParsedSentences Uses an optional

\* EntityMaintainer to work on a converted sentence.

\*/

private def parseSentence(\_sentence: String, entityMaintainer: EntityMaintainer): Sentence = {

var sentence = \_sentence

if (entityMaintainer != null) {

entityMaintainer.convertSentence(sentence, null)

sentence = entityMaintainer.getConvertedSentence

}

if (sentence == null) return null

val orig\_sentence = entityMaintainer.getOriginalSentence

var sent: Sentence = null

if (sentence.length < DEFAULT\_MAX\_SENTENCE\_LENGTH) {

sent = parser.parse(sentence)

}

else {

log.error("Sentence too long!: " + sentence)

sent = new Sentence()

}

sent.setSentence(orig\_sentence)

sent

}

private def reportTime(msg: String) {

val now: Long = System.currentTimeMillis

val elapsed: Long = now - \_starttime

\_starttime = now

var sum: Long = sumtime.get(msg)

var cnt: Long = cnttime.get(msg)

if (sum == null) {

sum = 0L

cnt = 0L

}

cnt += 1

sum += elapsed

sumtime.put(msg, sum)

cnttime.put(msg, cnt)

val avg: Long = sum / cnt

log.debug("KB " + msg + elapsed + " milliseconds (avg=" + avg + " millisecs, cnt=" + cnt + ")")

}

}

# Модуль coreservice.thinkinglifecycle

package tu.coreservice.thinkinglifecycle

import tu.model.knowledge.training.Goal

import tu.dataservice.knowledgebaseserver.KBAdapter

import tu.model.knowledge.communication.ShortTermMemory

/\*\*

\* @author max talanov

\* date 2012-07-18

\* time: 8:35 PM

\*/

class GoalManager {

private val goals = KBAdapter.workflow

private val trainingGoals = KBAdapter.trainingGoal.keys.toList

private var currentIndex = 0

/\*\*

\* Gets next to specified goal.

\* @param currentGoal Goal to process.

\* @return next Goal.

\*/

def nextGoal(currentGoal: Goal): Option[Goal] = {

if (!goals.contains(currentGoal)) {

None

} else {

val index = goals.indexOf(currentGoal) + 1

if (index > goals.size) {

None

} else {

Some(goals(index))

}

}

}

def nextGoal(inputContext: ShortTermMemory): Option[Goal] = {

inputContext.nextGoal match {

case Some(g: Goal) => Some(g)

case None => {

nextGoal

}

}

}

def nextTrainingGoal(inputContext: ShortTermMemory): Option[Goal] = {

inputContext.nextGoal match {

case Some(g: Goal) => Some(g)

case None => {

nextTrainingGoal

}

}

}

def nextTrainingGoal: Option[Goal] = {

this.currentIndex += 1

if (this.currentIndex >= goals.size) {

None

} else {

Some(trainingGoals(currentIndex))

}

}

def nextGoal: Option[Goal] = {

this.currentIndex += 1

if (this.currentIndex >= goals.size) {

None

} else {

Some(goals(currentIndex))

}

}

def resetIndex: Int = {

this.currentIndex = 0

this.currentIndex

}

def currentGoal: Option[Goal] = {

if (currentIndex >= goals.size || currentIndex < 0) {

None

} else {

Some(goals(this.currentIndex))

}

}

def currentTrainingGoal: Option[Goal] = {

if (currentIndex >= trainingGoals.size || currentIndex < 0) {

None

} else {

Some(trainingGoals(this.currentIndex))

}

}

}

package tu.coreservice.thinkinglifecycle

import akka.actor.{ActorRef, ActorDSL, ActorSystem}

import akka.util.Timeout

import tu.model.knowledge.communication.{ShortTermMemory, ContextHelper}

import tu.coreservice.action.{Action, ActionActor}

import tu.coreservice.action.event.{Stop, Start}

import org.slf4j.LoggerFactory

import akka.pattern.\_

import scala.concurrent.Await

/\*\*

\* Class to process parallel Actions and join results.

\* @author max talanov

\* date 2012-07-10

\* time: 3:20 PM

\*/

object JoinProcessor {

val log = LoggerFactory.getLogger(this.getClass)

implicit val system = ActorSystem("test-actor");

implicit val timeout = Timeout.longToTimeout(120\*60\*1000);

def apply(actions: List[Action], context: ShortTermMemory): ShortTermMemory = {

// initialisation and asynchronous call

val actionActors: List[ActorRef] = for (a <- actions) yield {

val aA =ActorDSL.actor(new ActionActor)

aA ! Start(a, context)

aA

}

// join

val contexts = for (a <- actionActors) yield {

val vl = a ? Stop;

val vl2= Await.result(vl,timeout.duration).asInstanceOf[ShortTermMemory]

vl2 match {

case res: ShortTermMemory =>

log info res.toString

res

}

}

ContextHelper.merge(contexts)

}

}

package tu.coreservice.thinkinglifecycle

import tu.coreservice.action.way2think.Way2Think

import tu.model.knowledge.communication.ShortTermMemory

import tu.coreservice.action.Action

/\*\*

\* @author max talanov

\* date 2012-07-12

\* time: 4:04 PM

\*/

case class JoinWay2Think(actions: List[Action]) extends Way2Think{

def start() = false

def stop() = false

/\*\*

\* Way2Think interface.

\* @param inputContext ShortTermMemory of all inbound parameters.

\* @return outputContext

\*/

def apply(inputContext: ShortTermMemory) = null

}

package tu.coreservice.thinkinglifecycle

import tu.model.knowledge.communication.{ShortTermMemory, Request}

/\*\*

\* Main component to manage Action-s (Selector, Way2Think,Critic).

\* ThinkingLifeCycle starts any Action as parallel process, except for the Way2Think grouped in Sequence.

\* All sequences are started as parallel processes.

\* Actions has levels and higher level actions manage low level actions.

\* @author max talanov

\* date 2012-07-07

\* time: 7:42 PM

\*/

trait ThinkingLifeCycle {

def apply(request: Request): ShortTermMemory

}

package tu.coreservice.thinkinglifecycle

import tu.model.knowledge.communication.\_

import tu.coreservice.action.selector.Selector

import tu.coreservice.action.Action

import tu.model.knowledge.Resource

import tu.model.knowledge.way2think.{JoinWay2ThinkModel, Way2ThinkModel}

import tu.coreservice.action.way2think.cry4help.Cry4HelpWay2Think

import tu.model.knowledge.critic.CriticModel

import tu.model.knowledge.action.ActionModel

import tu.model.knowledge.training.Goal

import tu.model.knowledge.selector.SelectorRequest

import org.slf4j.LoggerFactory

import tu.exception.UnexpectedException

import tu.dataservice.knowledgebaseserver.KBAdapter

import tu.dataservice.memory.LongTermMemory

import tu.model.knowledge.communication.ShortTermMemory

import scala.Some

import tu.model.knowledge.domain.ConceptNetwork

/\*\*

\* Minimal implementation of ThinkingLifeCycle.

\* @author max talanov

\* date 2012-07-11

\* time: 11:42 PM

\*/

class ThinkingLifeCycleMinimal

extends ThinkingLifeCycle {

val log = LoggerFactory.getLogger(this.getClass)

val selector = new Selector

var globalContext = ContextHelper(List[Resource](), "globalContext")

private def initializeGlobalContext(request: Request) = {

globalContext = ContextHelper(List[Resource](request.inputText), request.inputText, "globalContext")

globalContext.domainModel = LongTermMemory.domainModel(request.domainName)

globalContext.simulationModel = LongTermMemory.simulationModel(request.domainName)

globalContext.reformulationModel = LongTermMemory.reformulationModel(request.domainName)

globalContext.solutions = LongTermMemory.solutions(request.domainName)

}

/\*\*

\* Runs Goals linked to Request as work-flows.

\* @param request to process.

\*/

def apply(request: TrainingRequest): ShortTermMemory = {

log debug "apply(" + request + ": TrainingRequest))"

initializeGlobalContext(request)

val goalManager = new GoalManager

var resGoals: List[Goal] = List[Goal]()

// process resources

while (goalManager.currentTrainingGoal != None) {

// get next goal

// process next goal

val goalOption = goalManager.currentTrainingGoal

goalOption match {

case Some(goal: Goal) => {

log debug "Goal:" + goal

resGoals = resGoals ::: List(goal)

val resources: List[Resource] = selector.apply(goal)

val contexts = processResources(resources)

this.globalContext = mergeContexts(contexts)

val refContexts = processReflectiveCritics(globalContext)

this.globalContext = mergeContexts(refContexts)

log debug "out Contexts: " + contexts.toString()

}

case None => //End

}

goalManager.nextGoal(globalContext)

}

log debug "apply()"

globalContext.lastResult match {

case Some(r: ConceptNetwork) => {

globalContext.domainModel match {

case Some(model: ConceptNetwork) => {

model.nodes = model.nodes ::: r.nodes

globalContext.domainModel = Some(model)

LongTermMemory.saveModel(request.domainName, model)

}

case None => {

// no domain model => do nothing

}

}

}

case None => {

//no updated concepts => do nothing

}

}

globalContext

}

/\*\*

\* Runs Goals linked to Request as work-flows.

\* @param request to process.

\*/

def apply(request: Request): ShortTermMemory = {

log debug "apply(" + request + ": Request))"

initializeGlobalContext(request)

val goalManager = new GoalManager

var resGoals: List[Goal] = List[Goal]()

// get selector resources for request this is first goal = Goal("ProcessIncident")

// val resources: List[Resource] = selector.apply(request)

// currently all goals are in goals list in KBPrimitive

// process resources

while (goalManager.currentGoal != None) {

// get next goal

// process next goal

val goalOption = goalManager.currentGoal

goalOption match {

case Some(goal: Goal) => {

log debug "Goal:" + goal

resGoals = resGoals ::: List(goal)

val resources: List[Resource] = selector.apply(goal)

val contexts = processResources(resources)

this.globalContext = mergeContexts(contexts)

val refContexts = processReflectiveCritics(globalContext)

this.globalContext = mergeContexts(refContexts)

// log debug "out Contexts: " + contexts.toString()

}

case None => //End

}

goalManager.nextGoal(globalContext)

}

log debug "apply()"

globalContext

}

def mergeContexts(contexts: List[ShortTermMemory]): ShortTermMemory = {

val mergedRefContexts = ContextHelper.mergeLast(contexts)

this.globalContext = copyGlobalContext(mergedRefContexts)

this.globalContext

}

def processSelectorRequest(request: SelectorRequest): List[ShortTermMemory] = {

val resources: List[Resource] = selector.apply(request)

val contexts = processResources(resources, reflectiveFlag = true)

contexts

}

/\*\*

\* Runs through resources and interprets them as Critics or Way2Think with global context, stores result in global context.

\* @param resources to process.

\* @param reflectiveFlag Boolean flag to identify reflectiveResult use.

\* @return List of Contexts results of processing.

\*/

def processResources(resources: List[Resource], reflectiveFlag: Boolean = false): List[ShortTermMemory] = {

log trace "processResources(" + resources + ": List[Resource] " + reflectiveFlag + " Boolean ): List[ShortTermMemory]"

val contexts: List[List[ShortTermMemory]] = for (r <- resources) yield {

val resContext = translate(r, this.globalContext)

if (resContext != null) {

globalContext = copyGlobalContext(resContext)

}

val contextToCheck = if (!reflectiveFlag) {

(resContext.lastResult) match {

case Some(sR: SelectorRequest) => {

this.processSelectorRequest(sR)

}

case \_ => List[ShortTermMemory](resContext)

}

} else {

(resContext.lastReflectiveResult) match {

case Some(sR: SelectorRequest) => {

this.processSelectorRequest(sR)

}

case \_ => List[ShortTermMemory](resContext)

}

}

contextToCheck

}

log debug "processResources(): List[ShortTermMemory] = " + contexts.flatten.toString()

contexts.flatten

}

/\*\*

\* Start reflective critics and Cry4Help Way2Think

\* @param contextToCheck the context to check by reflective critics

\* @return ShortTermMemory with SelectorRequest-s

\*/

def processReflectiveCritics(contextToCheck: ShortTermMemory): List[ShortTermMemory] = {

val reflectiveCritics: List[CriticModel] = KBAdapter.getReflectiveCritics

processResources(reflectiveCritics, true)

}

def translate(resource: Resource, globalContext: ShortTermMemory): ShortTermMemory = {

log debug "translate(" + resource + ": Resource, " + globalContext + ": ShortTermMemory)"

resource match {

case joinWay2Think: JoinWay2ThinkModel => {

// run JoinProcessor

val parameters = joinWay2Think.parameters

val actions: List[Action] = parameters.map {

a: ActionModel => this.instantiate(a.uri.name)

}

val res = JoinProcessor(actions, globalContext)

log debug "translate(): ShortTermMemory " + res.toString

res

}

case w2t: Way2ThinkModel => {

val action = this.instantiate(w2t.uri.name)

val res = action.apply(globalContext)

log debug "translate(): ShortTermMemory " + res.toString

res

}

case critic: CriticModel => {

val action = this.instantiate(critic.uri.name)

val res = action.apply(globalContext)

log debug "translate(): ShortTermMemory " + res.toString

res

}

}

}

def instantiate(className: String): Action = {

log debug "instantiate(" + className + ": String): Action"

val clazz = Class.forName(className)

try {

val temp = clazz.newInstance()

val instance = temp.asInstanceOf[Action]

log debug "instantiate(): Action = " + instance.toString

instance

} catch {

case e: ClassCastException => {

Cry4HelpWay2Think("$Not\_alloved\_class " + clazz.getName)

throw new UnexpectedException("$Not\_alloved\_class " + clazz.getName)

}

}

}

def copyGlobalContext(resContext: ShortTermMemory): ShortTermMemory = {

this.globalContext = ContextHelper.mergeWithBaseContext(globalContext, List(this.globalContext, resContext))

this.globalContext

}

}

# Модуль dataservice.knowledgebaseserver

package tu.dataservice.knowledgebaseserver

import providers.N4JKB

import tu.model.knowledge.training.Goal

import tu.model.knowledge.way2think.{JoinWay2ThinkModel, Way2ThinkModel}

import tu.model.knowledge.action.ActionModel

import tu.model.knowledge.critic.CriticModel

import tu.model.knowledge.\_

import tu.model.knowledge.domain.{ConceptNetwork, Concept}

import tu.model.knowledge.annotator.AnnotatedPhrase

import tu.model.knowledge.howto.Solution

import tu.model.knowledge.Constant

/\*\*

\* KBSever stub only for prototype purposes.

\* @author max talanov

\* date 2012-07-06

\* time: 1:58 PM

\*/

object KBAdapter {

var kb = N4JKB

private def goalResourceMap =

Map[Goal, List[ActionModel]](

Goal("ProcessIncident") ->

List[Way2ThinkModel](Way2ThinkModel("tu.coreservice.splitter.PreliminarySplitter"),

Way2ThinkModel("tu.coreservice.annotator.KBAnnotatorImpl"),

Way2ThinkModel("tu.coreservice.linkparser.LinkParser")

),

Goal("ClassifyIncident") ->

List[JoinWay2ThinkModel](JoinWay2ThinkModel(

List[CriticModel](CriticModel("tu.coreservice.action.critic.analyser.DirectInstructionAnalyserCritic"),

CriticModel("tu.coreservice.action.critic.analyser.ProblemDescriptionAnalyserCritic"),

CriticModel("tu.coreservice.action.critic.analyser.ProblemDescriptionWithDesiredStateAnalyserCritic")

), "tu.model.knowledge.way2think.JoinWay2ThinkModel")

),

Goal("FormalizeDirectInstruction") ->

List[Way2ThinkModel](Way2ThinkModel("tu.coreservice.action.way2think.simulation.SimulationWay2Think")),

Goal("FormalizeProblemDescription") ->

List[Way2ThinkModel](Way2ThinkModel("tu.coreservice.action.way2think.simulation.SimulationWay2Think"),

Way2ThinkModel("tu.coreservice.action.way2think.reformulation.ReformulationWay2Think")),

Goal("GetMostProbableAction") ->

List[Way2ThinkModel](Way2ThinkModel("tu.coreservice.action.way2think.FindMostProbableAction")

),

Goal("SearchSolution") ->

List[Way2ThinkModel](Way2ThinkModel("tu.coreservice.action.way2think.SearchSolution")

),

Goal("ProcessResponse") ->

List[Way2ThinkModel](Way2ThinkModel("tu.coreservice.splitter.PreliminarySplitter"),

Way2ThinkModel("tu.coreservice.annotator.KBAnnotatorImpl"),

Way2ThinkModel("tu.coreservice.linkparser.LinkParser"),

Way2ThinkModel("tu.coreservice.action.way2think.simulation.CorrelationWay2Think")

),

Goal("Train") ->

List[Way2ThinkModel](Way2ThinkModel("tu.coreservice.splitter.PreliminarySplitter"),

Way2ThinkModel("tu.coreservice.annotator.KBAnnotatorImpl"),

Way2ThinkModel("tu.coreservice.linkparser.LinkParser"),

Way2ThinkModel("tu.coreservice.action.way2think.correlation.CorrelationWay2Think")

),

Goal("Cry4Help") ->

List[Way2ThinkModel](Way2ThinkModel("tu.coreservice.action.way2think.cry4help.Cry4HelpWay2Think")

)

)

private def resources = goalResourceMap.values

/\*\*

\* Gets Map of URI -> Resource of all registered Way2ThinkModel, CriticModel, JoinWay2ThinkModel

\* @return Map[KnowledgeURI, Resource]

\*/

private def uriResourcesMap: Map[KnowledgeURI, Resource] = {

val res: Map[KnowledgeURI, Resource] = goalResourceMap.values.flatten.map {

r: Resource => {

Pair(r.uri, r)

}

}.toMap

res

}

/\*\*

\* Gets Map of String -> Resource of all registered Way2ThinkModel, CriticModel, JoinWay2ThinkModel

\* @return Map[String, Resource]

\*/

def stringResourcesMap: Map[String, Resource] = {

val res: Map[String, Resource] = goalResourceMap.values.flatten.map {

r: Resource => {

Pair(r.uri.name, r)

}

}.toMap

res

}

def workflow = List(Goal("ProcessIncident"), Goal("ClassifyIncident"), Goal("GetMostProbableAction"), Goal("SearchSolution"))

def trainingGoal = Map[Goal, List[ActionModel]](

Goal("Train") ->

List[Way2ThinkModel](Way2ThinkModel("tu.coreservice.splitter.PreliminarySplitter"),

Way2ThinkModel("tu.coreservice.annotator.KBAnnotatorImpl"),

Way2ThinkModel("tu.coreservice.linkparser.LinkParser"),

Way2ThinkModel("tu.coreservice.action.way2think.correlation.CorrelationWay2Think")

)

)

def getByGoalName(name: String): Option[List[ActionModel]] = {

val resources = this.goalResourceMap

val keys: Iterable[Goal] = resources.keys.filter {

g: Goal => {

g.uri.name.equals(name)

}

}

if (keys.size > 0) {

resources.get(keys.head)

} else {

None

}

}

/\*\*

\* Clean the whole database and reinit

\*/

def cleanDatabase()

{

kb.clean()

}

def annotations: Map[String, AnnotatedPhrase] = Defaults.phrases.map(

(phrase: AnnotatedPhrase) => {

phrase.text -> phrase

}

).toMap

val uri = new KnowledgeURI("namespace", "name", "revision")

val probability = new Probability

//@deprecated

//def domainModel(): ConceptNetwork = someModel(Constant.domainName)

//def domainModel(name: String) = someModel(name)

def domainModel(uri: KnowledgeURI) = someModel(uri)

//@deprecated

//def simulationModel(): ConceptNetwork = someModel(Constant.simulationName)

//def simulationModel(name: String): ConceptNetwork = someModel(name)

def simulationModel(uri: KnowledgeURI): ConceptNetwork = someModel(uri)

//@deprecated

//def reformulationModel(): ConceptNetwork = someModel(Constant.reformulationName)

//def reformulationModel(name: String): ConceptNetwork = someModel(name)

def reformulationModel(uri: KnowledgeURI): ConceptNetwork = someModel(uri)

private def someModel(modelName: KnowledgeURI): ConceptNetwork = {

try {

ConceptNetwork.load(kb, KBNodeId(0), modelName.uri().get.toString, Constant.DEFAULT\_LINK\_NAME)

}

catch {

case \_ =>

val res: ConceptNetwork = Defaults.domainModelConceptNetwork

res.save(kb, KBNodeId(0), modelName.uri().get.toString, Constant.DEFAULT\_LINK\_NAME)

res

}

}

def solutions(): List[SolvedIssue] = {

val res: List[SolvedIssue] = kb.loadChildrenList(Constant.solutionsName).map(x => SolvedIssue.load(kb, x))

if (res.isEmpty) {

//save solutions

getDefaultSolutions

} else {

res

}

}

def solutionsAdd(item: SolvedIssue): List[SolvedIssue] = {

item.save(kb, KBNodeId(KB.getRootId()), item.uri.toString, Constant.solutionsName)

solutions()

}

private def getDefaultSolutions: List[SolvedIssue] = {

val in\_uri = new KnowledgeURI("namespace", "name", "revision")

val uri = new KnowledgeURI("namespace", "name", "revision")

def getTestSolvedIssue1: SolvedIssue = {

val s = new Solution(List(Defaults.generateInstallFirefoxHowTo), new KnowledgeURI("solutions", "InstallFirefoxSolution", "0.1"))

new SolvedIssue(Defaults.pleaseInstallFFSimulation, s, new KnowledgeURI("solutions", "InstallFirefoxSolvedIssue", "0.1"), new Probability)

}

def getTestSolvedIssue2: SolvedIssue = {

val s = new Solution(List(Defaults.generateReinstallIE8HowTo), new KnowledgeURI("solutions", "ReinstallIESolution", "0.1"))

new SolvedIssue(Defaults.iHaveProblemWithIE8Simulation, s, new KnowledgeURI("solutions", "ReinstallIESolvedIssue", "0.1"), probability)

}

def getTestSolvedIssue3: SolvedIssue = {

val s = new Solution(List(Defaults.generateReinstallIE8HowTo), new KnowledgeURI("solutions", "ReinstallIESolution", "0.1"))

new SolvedIssue(Defaults.iHaveProblemWithIE8Reformulation, s, new KnowledgeURI("solutions", "ReinstallIESolvedIssue", "0.1"), probability)

}

List(getTestSolvedIssue1, getTestSolvedIssue2, getTestSolvedIssue3)

}

/\*\* \*

\* Gets annotations according to specified word

\* @param word to find annotations

\* @return annotated phrase by word (for example get rid off)

\*/

def getAnnotationByWord(word: String): Option[AnnotatedPhrase] = {

var resources = kb.loadChildrenList(Constant.savedAnnotations).map(x => AnnotatedPhrase.load(kb, x))

if (resources.isEmpty) {

resources = Defaults.phrases

}

val phrases: Iterable[AnnotatedPhrase] = resources.toList.filter {

g: AnnotatedPhrase => {

g.toString.equals(word)

}

}

if (phrases.size > 0) {

Option(phrases.head)

} else {

None

}

}

def getReflectiveCritics: List[CriticModel] = {

var list = kb.loadChildrenList(Constant.selfReflectiveCritics).map(x => CriticModel.load(kb, x))

if (list.isEmpty) {

//save list to db and return

list = Defaults.defaultSelfReflectiveCritics

}

list

}

//object Defaults moved to InitialData file

}

package tu.dataservice.knowledgebaseserver

import tu.model.knowledge.annotator.AnnotatedPhrase

import tu.model.knowledge.howto.HowTo

import tu.model.knowledge.frame.Frame

import tu.model.knowledge.domain.{ConceptNetwork, ConceptTag, ConceptLink, Concept}

import tu.model.knowledge.{KnowledgeURI, Probability}

import tu.model.knowledge.critic.CriticModel

/\*\*

\* @author alex toschev

\* @author max talanov

\* Time stamp: 9/18/12 6:29 PM

\*/

object Defaults {

val CONCEPT = Concept("concept")

val CONCEPT\_LINK = ConceptLink(CONCEPT, CONCEPT, "conceptLink")

val conceptPhrase = AnnotatedPhrase("concept", CONCEPT)

val wordConcept = Concept.createSubConcept(CONCEPT, "word")

val wordPhrase = AnnotatedPhrase("word", wordConcept)

val subjectConcept = Concept.createSubConcept(CONCEPT, "subject")

val subjectPhrase = AnnotatedPhrase("subject", subjectConcept)

val objectConcept = Concept.createSubConcept(CONCEPT, "object")

val objectPhrase = AnnotatedPhrase("object", objectConcept)

val formOfPolitenessConcept = Concept.createSubConcept(CONCEPT, "FormOfPoliteness")

val formOfPolitenessPhrase = AnnotatedPhrase("form of politeness", formOfPolitenessConcept)

val actionConcept = Concept.createSubConcept(CONCEPT, "action")

val actionPhrase = AnnotatedPhrase("action", actionConcept)

val programConcept = Concept.createSubConcept(CONCEPT, "program")

val programPhrase = AnnotatedPhrase("program", programConcept)

val beConcept = Concept.createSubConcept(CONCEPT, "be")

val hasConcept = Concept.createSubConcept(CONCEPT, "has")

val consistConcept = Concept.createSubConcept(CONCEPT, "consistOf")

/\*\*

\* Links

\*/

val missLink = ConceptLink(userConcept, objectConcept, "miss")

val hasNo = ConceptLink(subjectConcept, objectConcept, "hasNo")

val appliedLink = ConceptLink(subjectConcept, objectConcept, "applied")

val has = ConceptLink.createSubConceptLink(CONCEPT\_LINK, subjectConcept, objectConcept, "has", new Probability(1.0, 1.0))

val hasPhrase = AnnotatedPhrase("has", has)

val beLink = ConceptLink.createSubConceptLink(CONCEPT\_LINK, subjectConcept, objectConcept, "be")

val isPhrase = AnnotatedPhrase("is", beLink)

val generalisationLink = ConceptLink.createSubConceptLink(CONCEPT\_LINK, subjectConcept, objectConcept, "generalisation")

val isAPhrase = AnnotatedPhrase("is a", generalisationLink)

val kindOfPhrase = AnnotatedPhrase("kind of", generalisationLink)

val missPhrase = AnnotatedPhrase("miss", missLink )

/\*\*

\* Link reduction

\*/

val reduceLinks = Map(beConcept -> beLink, hasConcept -> has, consistConcept -> has )

val objectLinkName = "obj"

val subjectLinkName = "subj"

val reductionConceptLinks = List(objectLinkName, subjectLinkName)

/\*\*

\* Containers

\*/

val concepts = List[Concept](CONCEPT, wordConcept, subjectConcept, objectConcept, formOfPolitenessConcept, actionConcept, programConcept, beConcept, hasConcept)

val conceptLinks: List[ConceptLink] = List(CONCEPT\_LINK, has, beLink, generalisationLink)

val phrases: List[AnnotatedPhrase] = List(conceptPhrase, wordPhrase, subjectPhrase, objectPhrase, hasPhrase, isPhrase, isAPhrase, kindOfPhrase, missPhrase, formOfPolitenessPhrase)

val softwareConcept = Concept.createSubConcept(objectConcept, "software")

val browserConcept = Concept.createSubConcept(softwareConcept, "Browser")

val internetExplorerConcept = Concept.createSubConcept(browserConcept, "Microsoft Internet Explorer")

val versionConcept = Concept.createSubConcept(objectConcept, "version")

val userConcept = Concept.createSubConcept(subjectConcept, "user")

val addressConcept = Concept.createSubConcept(objectConcept, "address")

val computerConcept = Concept.createSubConcept(objectConcept, "computer")

val firefoxConcept = Concept.createSubConcept(browserConcept, "Firefox")

val systemConcept = Concept.createSubConcept(objectConcept, "system")

// lexical

val tenseConcept = Concept("tense")

val posConcept = Concept("pos")

val formOfPoliteness = Concept("formOfPoliteness")

val tenseLink = ConceptLink(subjectConcept, objectConcept, "tense")

val posLink = ConceptLink(subjectConcept, objectConcept, "pos")

/\*\*

\* domain-s (ConceptNetworks

\*/

val domainModelConceptNetwork = ConceptNetwork(Defaults.concepts, Defaults.conceptLinks, KnowledgeURI("domainModel"))

/\*\*

\* HowTo-s

\*/

val installHowTo = new HowTo(List[Frame](Frame(CONCEPT)), List[ConceptTag](), KnowledgeURI("installHowTo"))

val reinstallHowTo = new HowTo(List[Frame](Frame(CONCEPT)), List[ConceptTag](), KnowledgeURI("reinstallHowTo"))

// actions

val installConcept = Concept.createSubConcept(actionConcept, "install")

val removeConcept = Concept.createSubConcept(actionConcept, "remove")

val cleanConcept = Concept.createSubConcept(actionConcept, "clean")

/\*\*

\* Generates reinstall IE8 HowTo

\* @return HowTo

\*/

def generateReinstallIE8HowTo = reinstallIEHowTo

/\*\*

\* Generates install Firefox HowTo

\* @return HowTo

\*/

def generateInstallFirefoxHowTo = installFirefoxHowTo

val installFirefoxHowTo = HowTo.createInstance(installHowTo, List(Frame(firefoxConcept)))

val reinstallIEHowTo = HowTo.createInstance(installHowTo, List(Frame(internetExplorerConcept)))

// User miss Internet Explorer 8 simulated

val userInst = Concept.createInstanceConcept(userConcept)

val internetExplorerInst = Concept.createInstanceConcept(internetExplorerConcept)

val versionInst = Concept.createInstanceConcept(versionConcept, "8")

val userMissInternetExplorer = ConceptLink.createInstanceConceptLink(missLink, userInst, internetExplorerInst)

val internetExplorerHasVersion = ConceptLink.createInstanceConceptLink(has, internetExplorerInst, versionInst)

val iHaveProblemWithIE8Simulation = new ConceptNetwork(List[Concept](userInst, internetExplorerInst, versionInst),

List[ConceptLink](userMissInternetExplorer, internetExplorerHasVersion), KnowledgeURI("iHaveProblemWithIE8Simulation"))

// User miss Internet Explorer 8 reformulated

val userInstRef = Concept.createInstanceConcept(userConcept)

val computerInstRef = Concept.createInstanceConcept(computerConcept)

val userHasComputerInst = ConceptLink.createInstanceConceptLink(has, userInstRef, computerInstRef)

val addressInstRef = Concept.createInstanceConcept(addressConcept, "someAddress")

val computerHasAddressRef = ConceptLink.createInstanceConceptLink(has, computerInstRef, addressInstRef)

val internetExplorerInstRef = Concept.createInstanceConcept(internetExplorerConcept)

val computerHasNoInternetExplorerInstRef = ConceptLink.createInstanceConceptLink(hasNo, computerInstRef, internetExplorerInstRef)

val internetExplorerHasVersionInstRef = ConceptLink.createInstanceConceptLink(has, internetExplorerInstRef, versionInst)

val iHaveProblemWithIE8Reformulation = new ConceptNetwork(List[Concept](userInstRef, computerInstRef, addressInstRef, internetExplorerInstRef),

List[ConceptLink](userHasComputerInst, computerHasAddressRef, computerHasNoInternetExplorerInstRef, internetExplorerHasVersionInstRef), KnowledgeURI("iHaveProblemWithIE8Reformulation"))

// Please install Firefox simulation

val installActionInst = Concept.createInstanceConcept(installConcept)

val firefoxConceptInst = Concept.createInstanceConcept(firefoxConcept)

val systemInstallFirefox = ConceptLink.createSubConceptLink(appliedLink, systemConcept, firefoxConceptInst, "systemInstallFirefox")

val pleaseInstallFFSimulation = new ConceptNetwork(List[Concept](installActionInst, firefoxConceptInst), List[ConceptLink](systemInstallFirefox),

KnowledgeURI("pleaseInstallFFSimulation"))

val defaultSelfReflectiveCritics = List(CriticModel("tu.coreservice.action.critic.manager.DoNotUnderstandManager"))

}

package tu.dataservice.knowledgebaseserver.providers

import org.neo4j.graphdb.{Relationship, Transaction, RelationshipType, Node}

import tu.model.knowledge.\_

import org.slf4j.LoggerFactory

import com.typesafe.config.ConfigFactory

import org.neo4j.kernel.EmbeddedGraphDatabase

import collection.mutable

import collection.immutable.HashMap

import org.neo4j.graphdb.index.Index

import org.neo4j.cypher.commands.Return

import java.io.File

import tools.nsc.io.Directory

/\*\*

\*

\* @author Alexander Toschev

\* Date: 10/30/12

\* Time: 5:51 AM

\*

\*/

object N4JKB extends KB {

val log = LoggerFactory.getLogger(this.getClass)

val defaultFilename = {

//load file name or use default

val conf = ConfigFactory.load()

val suffix = conf.getString("tu.knowledgebaseserver.dir")

if (suffix.size == 0)

java.lang.System.getProperty("user.home") + "/" + "tu\_kb"

else if (suffix.charAt(0) == '/' || suffix.charAt(2) == ':')

suffix

else

java.lang.System.getProperty("user.home") + "/" + suffix

}

val keyField = "key"

private var inited = false

private var \_GraphDb: EmbeddedGraphDatabase = \_

private var \_nodeIndex: IndexedSeq[Node] = \_

def apply(): EmbeddedGraphDatabase = {

if (!inited) {

\_GraphDb = new EmbeddedGraphDatabase(defaultFilename)

log.trace("Neo4j database initialized. Location "+defaultFilename)

ShutdownHook(\_GraphDb.shutdown())

//TODO: \_nodeIndex = \_GraphDb.index().forNodes( "nodes" );

inited = true

}

\_GraphDb

}

override def saveResource(resource: Resource, key: String, linkType: String): Boolean = {

saveResource(resource, N4JKB().getReferenceNode, key, linkType)

}

override def loadChild(key: String, linkType: String): Map[String, String] = loadChild(N4JKB().getReferenceNode, key, linkType)

override def loadChildrenList(linkType: String): List[Map[String, String]] = loadChildrenList(N4JKB().getReferenceNode, linkType)

override def loadChildrenMap(linkType: String): Map[String, Map[String, String]] = loadChildrenMap(N4JKB().getReferenceNode, linkType)

override def saveResource(child: Resource, parent: KBNodeId, key: String, linkType: String = "defaultLink"): Boolean = {

saveResource(child, getNodeById(parent.ID), key, linkType)

}

override def loadChild(parent: KBNodeId, key: String, linkType: String): Map[String, String] = loadChild(getNodeById(parent.ID), key, linkType)

override def loadChildrenList(parent: KBNodeId, linkType: String): List[Map[String, String]] = loadChildrenList(getNodeById(parent.ID), linkType)

override def loadChildrenMap(parent: KBNodeId, linkType: String): Map[String, Map[String, String]] = loadChildrenMap(getNodeById(parent.ID), linkType)

private def getNodeById(Id: Long): Node = {

try {

return N4JKB().getNodeById(Id)

}

catch {

case \_ => LoggerFactory.getLogger(this.getClass).error("Attempt to get not existing node with ID {}", Id.toString)

}

N4JKB().getReferenceNode

}

def createLink(parent: KBNodeId, child:KBNodeId,linkType: String, key:String)={

var ok = false

val tx: Transaction = N4JKB().beginTx()

try {

val parentNode = N4JKB().getNodeById(parent.ID)

val childNode = N4JKB().getNodeById(child.ID)

val relationship = parentNode.createRelationshipTo(childNode,new RelationType(linkType))

relationship.setProperty("key", key)

tx.success()

ok = true

}

finally {

tx.finish()

}

ok

}

private def saveResource(child: Resource, parentNode: Node, key: String, linkType: String): Boolean = {

val relationType = new RelationType(linkType)

var ok = false

val tx: Transaction = N4JKB().beginTx()

try {

val childNode = N4JKB().createNode()

for ((x, y) <- child.export)

childNode.setProperty(x, y)

childNode.setProperty(Constant.KB\_CLASS\_NAME, child.getClass.getName)

val relationship = parentNode.createRelationshipTo(childNode, relationType)

relationship.setProperty("key", key)

tx.success()

KBMap.register(child, childNode.getId)

ok = true

}

finally {

tx.finish()

}

ok

}

private def loadChild(parent: Node, key: String, linkType: String): Map[String, String] = {

val relationType = new RelationType(linkType)

//TODO use кошерныйсинтаксис

val i = parent.getRelationships(relationType).iterator()

while (i.hasNext) //(x:Relationship <- parent.getRelationships.iterator())

{

val relationship: Relationship = i.next()

if (relationship.getProperty("key") == key) {

val node: Node = relationship.getEndNode

var values = Map[String, String]()

val j = node.getPropertyKeys.iterator()

while (j.hasNext) {

val key: String = j.next()

values += key -> node.getProperty(key).toString

}

values += (Constant.KB\_ID -> node.getId.toString)

return values

}

}

Nil.toMap[String, String]

}

private def loadChildrenList(parent: Node, linkType: String): List[Map[String, String]] = {

val mapChild = loadChildrenMap(parent, linkType)

mapChild.values.toList

}

private def loadChildrenMap(parent: Node, linkType: String): Map[String, Map[String, String]] = {

val relationType = new RelationType(linkType)

var res = new HashMap[String, Map[String, String]]

//TODO use кошерныйсинтаксис

val i = parent.getRelationships(relationType).iterator()

while (i.hasNext) //(x:Relationship <- parent.getRelationships.iterator()) {

{ val relationship: Relationship = i.next()

val node: Node = relationship.getEndNode

if (node.getId != parent.getId) {

var values = new HashMap[String, String]

val j = node.getPropertyKeys.iterator()

while (j.hasNext) {

val key: String = j.next()

values += key -> node.getProperty(key).toString

}

values += (Constant.KB\_ID -> node.getId.toString)

res += relationship.getProperty("key").toString -> values

}

}

res

}

/\*

override def loadSelf(selfID:KBNodeId):Map[String, String] ={

loadSelf(N4JKB().getNodeById(selfID))

}

private def loadSelf(node:Node):Map[String, String] = {

var values = new HashMap[String, String]

val j = node.getPropertyKeys.iterator()

while(j.hasNext)

{

val key:String = j.next()

values += key -> node.getProperty(key).toString

}

values += (Constant.KB\_ID -> node.getId.toString)

values

}

\*/

// private var goalIndex:Index[Node] = \_GraphDb.index().forNodes( Goal.getClass.getName )

def init(): Boolean = {

try {

N4JKB.apply()

true

}

catch {

case e: Exception => {

LoggerFactory.getLogger(this.getClass).error("Cann't create database: {}", e.toString)

}

false

}

}

/\*\*

\* Clean database . Use only for testing purpose.

\*/

def clean() {

if (\_GraphDb==null) N4JKB.apply()

\_GraphDb.shutdown()

log.debug("Cleaning directory: "+defaultFilename)

deleteFileOrDirectory(new File(defaultFilename))

log.debug("Reinit database: "+defaultFilename)

new File(defaultFilename).mkdirs()

N4JKB.apply()

}

def deleteFileOrDirectory( file:File ) {

if ( file.exists() ) {

if ( file.isDirectory() ) {

file.listFiles().foreach (f=>{

deleteFileOrDirectory (f)

})

}

file.delete();

}

}

//def goals = {goalIndex.ensuring(true)}

//List(Goal("ProcessIncident"), Goal("ClassifyIncident"), Goal("GetMostProbableAction"), Goal("SearchSolution"))

private def addIndexedNode(resource: Resource, index: Index[Node]): Option[Node] = addIndexedNode(resource, resource.uri.name, index)

private def addIndexedNode(resource: Resource, key: String, index: Index[Node]): Option[Node] = {

val tx: Transaction = \_GraphDb.beginTx()

try {

val node: Node = \_GraphDb.createNode()

node.setProperty("type", resource.getClass.getName)

node.setProperty("name", resource.uri.name)

index.add(node, "key", key)

tx.success()

return Option(node)

}

finally {

tx.finish()

}

None

}

def addIndexedNode(key: String): Node = {

val node: Node = \_GraphDb.createNode()

node.setProperty(keyField, key)

//TODO: \_nodeIndex.add( node, keyField , key );

node

}

}

class RelationType(\_name: String) extends RelationshipType {

def name(): String = {

\_name

}

}

/\*\*

\* A virtual machine shutdown hook.

\*

\* Registered shutdown hooks are executed when the virtual machine

\* terminates normally. For more information read the documentation

\* of Runtime. Constructor creates an unregistered instance

\* @param name the shutdown hook's name (for e.g. logging purposes)

\* @param body code to execute when the shutdown hook is executed

\*/

final class ShutdownHook(

val name: String,

private[this] val body: => Unit) {

def this(body: => Unit) = this("", body)

private[this] val hook = new Thread(new Runnable() {

def run {

body

}

}, name)

/\*\*Registers this as a virtual machine shutdown hook. \*/

def register() = {

Runtime.getRuntime().addShutdownHook(hook)

this

}

/\*\*Deregisters this from the virtual machine. \*/

def deregister() = Runtime.getRuntime.removeShutdownHook(hook)

override def toString = "ShutdownHook(" + name + ")"

}

/\*\*Factory for ShutdownHook instances. \*/

final object ShutdownHook {

/\*\*Creates a registered instance. \*/

def apply(name: String, body: => Unit) = new ShutdownHook(name, body).register()

/\*\*Creates a registered instance. \*/

def apply(body: => Unit) = new ShutdownHook("", body).register()

}

package tu.dataservice.knowledgebaseserver.relathionshipType

import org.neo4j.graphdb.RelationshipType

import tu.model.knowledge.way2think.Way2ThinkModel

/\*\*

\* Created with IntelliJ IDEA.

\* User: ChepkunovA

\* Date: 17.08.12

\* Time: 17:48

\* To change this template use File | Settings | File Templates.

\*/

class Has extends RelationshipType {

def name(): String = "HasRelationship"

}

class Goal extends RelationshipType {

def name(): String = {

tu.model.knowledge.training.Goal.getClass.getName

}

}

/\*

I hope, we should not create new type if it no need. "Has" relationship enough for most non-root nodes

class Way2ThinkModel extends RelationshipType

{

def name():String = {Way2ThinkModel.getClass.getName}

}

\*/

# Модуль model.knowledge

package tu.model.knowledge

/\*\*

\* Stores typed KLine

\* @author talanov max

\* date 2012-06-08

\* time: 10:51 PM

\* @see KLine

\*/

case class TypedKLine[Type <: Resource](var \_frames: Map[KnowledgeURI, Type], \_uri: KnowledgeURI, \_probability: Probability)

extends Resource(\_uri, \_probability) {

def size = frames.size

def this(\_frames: Map[KnowledgeURI, Type], \_uri: KnowledgeURI) = {

this(\_frames, \_uri, new Probability())

}

def frames: Map[KnowledgeURI, Type] = \_frames

def frames\_=(value: Map[KnowledgeURI, Type]): TypedKLine[Type] = {

\_frames = value

this

}

def +(in: Pair[KnowledgeURI, Type]): TypedKLine[Type] = {

\_frames = \_frames + in

this

}

def +(in: Type): TypedKLine[Type] = {

\_frames = \_frames + (in.uri -> in)

this

}

/\*\*

\* Returns Iterable of values of Type.

\* @return Iterable of values of Type.

\*/

def values: Iterable[Type] = frames.values

/\*\*

\* Returns Some[Type] if frames contains Resource with specified KnowledgeURI.

\* @param uri to search resource with.

\* @return Option[Type] with specified KnowledgeURI.

\*/

def get(uri: KnowledgeURI): Option[Type] = {

\_frames.get(uri)

}

/\*\*

\* Searches for Resource with KnowledgeURI with specified UID.

\* @param uid UID to search with.

\* @return Option[Type] with specified UID in KnowledgeURI.

\*/

def get(uid: String): List[Type] = {

\_frames.filter {

keyValue: Pair[KnowledgeURI, Type] => {

keyValue.\_1.uid.equals(uid)

}

}.map {

keyValue: Pair[KnowledgeURI, Type] => keyValue.\_2

}.toList

}

override def toString: String = {

frames.map(f => f.\_1.toString()).mkString("(", ",", ")")

}

//TODO correct this

/\*def this(map: Map[String, String]) = {

val typeString = map.get("type") match {

case Some(x) => x

case None => throw new UnexpectedException("$Type\_not\_specified")

}

val frames = map.get("frames") match {

case Some(x) => {

x

}

case None => Map[KnowledgeURI, Resource]()

}

this(Map[KnowledgeURI, Resource](), new KnowledgeURI(map), new Probability(map))

}\*/

/\*\*

\* Merges the frames of current TypedKLine with specified TypedKLine of the same Type.

\* @param in KLine to merge.

\* @return current TypedKLine with merged \_frames.

\*/

def merge(in: TypedKLine[Type]): TypedKLine[Type] = {

\_frames = frames ++ in.frames

this

}

}

object TypedKLine {

//TODO correct this

/\*

def translateStringMap[Type <: Resource](stringMap: String, typeString: String): Map[KnowledgeURI, Type] = {

val res = Map.empty[KnowledgeURI, Type]

val listPairsString: List[String] = stringMap.replaceFirst("Map\(", "").replace(")", "").split(",").toList

val listPairs: List[Pair[String, String]] = listPairsString.map {

x: String => {

val twoString = x.split("=>").toList

if (twoString.size > 1) {

(twoString(0), twoString(1))

} else {

throw new UnexpectedException("$Invalid\_map")

}

}

}

/\*listPairs.map {

}\*/

null

} \*/

def apply[Type <: Resource](uri: KnowledgeURI): TypedKLine[Type] = {

new TypedKLine(Map[KnowledgeURI, Type](), uri)

}

def apply[Type <: Resource](name: String): TypedKLine[Type] = {

new TypedKLine(Map[KnowledgeURI, Type](), KnowledgeURI(name + "TypedKLine"))

}

def apply[Type <: Resource](name: String, entity: Type): TypedKLine[Type] = {

new TypedKLine(Map[KnowledgeURI, Type](entity.uri -> entity), KnowledgeURI(name + "TypedKLine"))

}

def apply[Type <: Resource](name: String, entities: Map[KnowledgeURI, Type]): TypedKLine[Type] = {

new TypedKLine(entities, KnowledgeURI(name + "TypedKLine"))

}

}

package tu.model.knowledge

import Constant.DEFAULT\_LINK\_NAME

/\*\*

\* @author ${user.name}

\*/

trait KB {

//any objects stored with link from other (parent) object. If parent object not specified, then root node used insted one.

//

//to store simple object (node) into KnowledgeBase you should specify parent node, key of link ant type of link.

//

//to store complex object, you should store core of this object then store this components as specific links use this object as parent node

//

//to read one object, you should specify parent, key of relation and type of relation. first object with this key and type of relation will be read.

// if object has sub-objects, they should be read after

//

// to read set of objects you should specify only parent node and type of link. it is useful for read lists of sub-object.

// For example for store all generalisations of concept, used Constant.GENERALISATION\_LINK\_NAME

//

//

//common way of using database:

//

// to store

// Any object that should be stored, should contain "save(kb: KB, parent: KBNodeId, key: String, linkType: String)" method.

// If object has some children, it must invoke save-method and specify "this" as a parent argument.

// If object could be stored as child of root node, it should contain "save(kb: KB, key: String, linkType: String)" method.

// Inside save-methods described above KB.saveResource method with or without "parent" parameter should be invoked.

//

// to restore

// Object (helper) should contain load(kb: KB, parent: KBNodeId, key: String, linkType: String) method to load this node and its child nodes

//

// KBNodeId should be initialised from Resource of from Long or from map[String String]

// after object is restored fr0om database it can be registered by "register(r:Resource, id:Long)" method where r - loaded resource, ID its ID in KB

//

// If object has some simple fields, it should has constructor from Map[String, String] and export function which return appropriate Map[String, String]

// TODO edit and remove objects

def getIdFromMap(map:Map[String, String]) = KB.getIdFromMap(map)

// methods above should be implemented in real database server

// with root as parent

def saveResource(resource:Resource, key:String): Boolean = saveResource(resource, key, DEFAULT\_LINK\_NAME)

def saveResource(resource:Resource, key:String, linkType:String): Boolean = false

def loadChild(key:String):Map[String, String] = loadChild(key, DEFAULT\_LINK\_NAME)

def loadChild(key:String, linkType:String):Map[String, String] = Map()

def loadChildrenList():List[Map[String, String]] = loadChildrenList(DEFAULT\_LINK\_NAME)

def loadChildrenList(linkType:String):List[Map[String, String]] = List()

def loadChildrenMap():Map[String, Map[String, String]] = loadChildrenMap(DEFAULT\_LINK\_NAME)

def loadChildrenMap(linkType:String):Map[String, Map[String, String]] = Map()

// with Resource as parent

/\*\*

\* saves specified resource into KB as child of specified parent

\* @param child resource to be saved

\* @param parent parent of resource to be saved

\* @param nameOfLink name of link from parent to child/ Can be used to read specified child

\* @param linkType is defined by child type (for example geneneralisation and specialisation, ConceptList)

\* @return result of saving

\*/

def saveResource(child:Resource, parent:KBNodeId, nameOfLink:String, linkType:String = DEFAULT\_LINK\_NAME): Boolean = false

def loadChild(parent:KBNodeId, key:String, linkType:String = DEFAULT\_LINK\_NAME):Map[String, String] = Map()

def loadChildrenList(parent:KBNodeId, linkType:String = DEFAULT\_LINK\_NAME):List[Map[String, String]] = List()

def loadChildrenMap(parent:KBNodeId, linkType:String = DEFAULT\_LINK\_NAME):Map[String, Map[String, String]] = Map()

//def loadSelf(parent:KBNodeId):Map[String, String] = Map()

/\*\*

\* Create link between 2 nodes

\* @param parent reference

\* @param child reference

\* @param linkType type of link

\*/

def createLink(parent:KBNodeId, child:KBNodeId,linkType: String, key:String)

}

object KB {

def getRootId() = 0L

def getIdFromMap(map:Map[String, String]) = map.get(Constant.KB\_ID)

match {

case Some(x) => x.toLong

case None => Constant.NO\_KB\_NODE

}

}