SharkNet

September 2, 2015

Thilo Stegemann

1 Profile

Peers like Alice and Bob sometimes have more relevant information about themselves. If there is more than temporal interests an instance to store and collect theses relevant facts about peers is needed. Therefore we can use the profile. A profile can change its shape so it fits the needs. E.g. its possible to make an entry "ProfileName" and these entry can store a first name, a last name and a nick name but the entry can easily be changed so if there is no need for a nick name it can be changed into alternate spellings for the first name or add several new components to the "ProfileName" like academical titles and so on. An entry is a generic component of a profile. A profile can handle/manage several different entries. Content stored in an profile entry is persistent.

1.1 ProfileFactory

For creating/rebuilding a profile you always use the profile factory. These factory is like the manager of all profiles in the SharkKB. It provides functions to rebuild profiles out of context points and a function to create an empty new profile object `createProfile(PeerSemanticTag, PeerSemanticTag)`. The first PeerSemanticTag parameter is called creator and the second PeerSementicTag parameter is called target. Creator is a peer that makes a new profile and maintains it and a target is a peer all information of the profile refers to. E.g. a profile creator writes a profile over a target. A profile

factory needs to be initialized with a SharkKB. All profiles that are created with this factory are now stored in the KB. But the factory can only rebuild profiles from this specific SharkKB so if there is no such profile in the KB an SharkKBException is thrown. To rebuild a profile from an other KB create a new profile factory and initialize it with the KB.

1.1.1 Creating a profile

```
public class CreateProfile {

    //Create an empty SharkKB
    private SharkKB kb = new InMemoSharkKB();

    //Create a profile factory
    private ProfileFactory profileFactory;

    //Create Alice addresses
    private String[] aliceAddresses = {"mail://alice@wonderland.net",
      "mail://alice@wizards.net",
      "http://www.sharksystem.net/alice.html"};

    //Create Alice semantic identifiers
    private String[] aliceSis = {"http://www.sharksystem.net/alice.html"};

    //Create Alice as peer
    private PeerSemanticTag alice =
      InMemoSharkKB.createInMemoPeerSemanticTag("Alice",
      aliceSis,
      aliceAddresses);

    //Initialize profileFactory with SharkKB
    private CreateProfile() throws SharkKBException {
        profileFactory = new ProfileFactoryImpl(kb);
    }

    public Profile createProfileAlice() throws SharkKBException {
      //Create profile for Alice with the profileFactory
      Profile aliceProfile = profileFactory.createProfile(alice, alice);
```

```
        //Create ProfileName object
        ProfileName profileName = new ProfileNameImpl("Alice");

        //Fill ProfileName with data
        profileName.setLastName("Alpha");
        profileName.setTitle("Prof.");

        //Set the ProfileName in Alice profile
        aliceProfile.setName(profileName);
        return aliceProfile;
}
```

In these class a profile factory is used to create an profile for Alice. At first there is an empty SharkKB created which initialized the profileFactory, than a peer for Alice is created. This peer is needed because the createProfile function of the profileFactory needs peers as parameters. "aliceProfile" and a "profileName" object are created. The "profileName" object was filled with information about Alice first name, last name and titles of Alice. "setName" is called on "aliceProfile" and sets the "profileName" object persistently for Alice profile. A profile consists of all the entry functionality and some basic functions. These basic functions store data which often occurs in a normal profile like:

```
setName(ProfileName profileName)
setPicture(byte[] content, String contentType, String identifier)
setTelephoneNumber(String number, String identifier)
```

Just 3 basic functions to store often occurring data. It is possible to create exactly these 3 basic functions of the profile with the entry functionality but in terms of simplification they are given by the profile.

1.1.2 Getting profiles

```
public class GettingProfiles {
    public static void main(String[] args) throws SharkKBException {
      //Create empty SharkKB
      SharkKB kb = new InMemoSharkKB();
      ProfileFactory profileFactory = new ProfileFactoryImpl(kb);
```

```
        //Create a peer semantic tag describing Alice herself
        String[] aliceAddresses = {"mail://alice@wonderland.net",
        "mail://alice@wizards.net", "http://www.sharksystem.net/alice.html"};
        String[] aliceSis = {"http://www.sharksystem.net/alice.html"};
        PeerSemanticTag alice =
        InMemoSharkKB.createInMemoPeerSemanticTag("Alice",
                                aliceSis, aliceAddresses);

        //Create a peer semantic tag describing Bob himself
        String[] bobAddresses = {"mail://bob@sharknet.net",
        "mail://bob@wizards.net", "http://www.sharksystem.net/bob.html"};

        String[] bobSis = {"http://www.sharksystem.net/bob.html"};
        PeerSemanticTag bob = InMemoSharkKB.createInMemoPeerSemanticTag("Bob",
                                                        bobSis, bobAddresses);

        //Profile is created where Alice is the creator and the target
        Profile aliceProfile = profileFactory.createProfile(alice, alice);

        //Profile is created where Bob is the creator and the target
        Profile bobProfile = profileFactory.createProfile(bob, bob);

        //Create a list with Alice and Bobs profile in it
        List<profile> aliceAndBobProfiles = profileFactory.getAllProfiles();

        //Getting Alice profile
        Profile aliceNew1 = profileFactory.getProfile(alice);
        Profile aliceNew2 = profileFactory.getProfile(alice, alice);

        //aliceNew1 and aliceNew2 are equal profiles of Alice
    }
}
```

The first part is very similar to the example CreateProfile, because before extracting or getting profiles from the factory they must be created and in this example an empty SharkKB is used so it needs to be filled with profiles before. After the create section the SharkKB in the profileFactory now holds

a profile for Bob and a profile for Alice. These profiles can be extracted by using these functions:

- `List<Profile> getAllProfiles()`
- `Profile getProfile(PeerSemanticTag creatorAndTarget)`
- `Profile getProfile(PeerSemanticTag creator, PeerSemanticTag target)`

The first function returns a list with all profiles in the given profile factory, the second function needs a PeerSementicTag as parameter and will return a profile where the given peer is creator and target and the last function needs two PeerSemanticTags as parameters and will return a specific profile where the first peer of the profile is creator and the second peer of the profile is target.

1.2 Entry

An entry is an approach to be a generic container for information. Different kinds of information should be stored in entries. So the structure of entries is also alterable. Entry Example: Imagine an entry like a job employment it is part of a super entry WORK. A job employment can contain things like:

- employer name(String)
- job title(String)
- start and end of the job(Date or Int)
- information if the job is the current job(Boolean)
- job description(String)

These 5 together are now called an EMPLOYMENT. The super entry WORK consist of 3 sub entries:

- OCCUPATION
- SKILLS
- EMPLOYMENTS

1.2.1 Create WORK entry

```
//Interface that contains String constants as identifiers for entries
public interface CreateWorkEntry {
    String WORK = "Work";
    String OCCUPATION = "Occupation";
    String SKILLS = "Skills";
    String EMPLOYMENTS = "Employments";
    String EMPLOYERNAME = "EmployerName";
    String JOBTITLE = "JobTitle";
    String START = "Start";
    String END = "End";
    String CURRENT = "Current";
    String JOBDESCRIPTION = "JobDescription";

    void addEmployment(String employerName, String jobTitle,
            int start, int end, boolean current,
            String jobDescription) throws SharkKBException;

    Profile getProfileFromCreateWork();
}

public class CreateWorkEntryImpl implements CreateWorkEntry {

      //private member variable profile
      private Profile p;

      //The constructor needs an profile as parameter
      //e.g. aliceProfile from class CreateProfile
      CreateWorkEntryImpl(Profile p)  throws SharkKBException {
        this.p = p;
        createWork();
      }

      //Create general structure of WORK with no content
      private void createWork() throws SharkKBException {
        p.createProfileEntry(WORK);
        p.addSubEntryInEntry(WORK, OCCUPATION, "");
```

```
        p.addSubEntryInEntry(WORK, SKILLS, "");
        p.addSubEntryInEntry(WORK, EMPLOYMENTS);
      }

    //Creates EMPLOYMENT entries and consecutively numbered them
    public void addEmployment(String employerName, String jobTitle,
            int start, int end, boolean current,
            String jobDescription) throws SharkKBException {

        //A list of entries is created with parameters as content
        List<Entry<?>> entryList = new ArrayList<Entry<?>>();
        entryList.add(new EntryImpl<String>(EMPLOYERNAME, employerName));
        entryList.add(new EntryImpl<String>(JOBTITLE, jobTitle));
        entryList.add(new EntryImpl<Integer>(START, start));
        entryList.add(new EntryImpl<Integer>(END, end));
        entryList.add(new EntryImpl<Boolean>(CURRENT, current));
        entryList.add(new EntryImpl<String>(JOBDESCRIPTION, jobDescription));

        //Producing the consecutively numeration
        int count = p.getSubEntry(WORK, EMPLOYMENTS).getEntryList().size();

        //Sets the content "entryList"
        //with a number as name under entry EMPLOYMENTS
        p.createSubEntry(WORK, EMPLOYMENTS,
             Integer.toString(count), entryList);
    }

    public Profile getProfileFromCreateWork() {
      return p;
    }

}
```

CreateWorkEntry is an interface that maintains the head of all public functions in CreateWorkEntryImpl and all constants that are used to identify entries. Every entry needs an identifier like every human needs a name. Identifiers are used to add content, find entries or simply identify entries. To avoid spelling mistakes all identifiers are collected as constants in Create-

WorkEntry. Now entry names can be accessed by typing just wor... or employ... and intelligent IDEs will auto complete them. Without the interface constants its needed to write "WORK" or "EMPLOYMENTS" every time and if there is a spelling mistake the whole program crashes and exceptions are thrown. The class CreateWorkEntyImpl implements interface CreateWorkEntry to implement functions that are given in CreateWorkEntry and to use the constant identifiers for entries. CreateWorkEntryImpl has just one member variable "Profile p". These member is initialized in the constructor. A profile from outside is needed e.g. aliceProfile or bobProfile. In the end they will get the entry WORK and if the function `addEmployment(...)` is used the first EMPLOYMENT entry is created. The access of EMPLOYMENT content is explained in a later section. `addEmployment(...)` first creates a list of entries then it adds every single needed entry with content in the list. After the list is completed it counts how many items the entry EMPLOYMENTS has. The count is increased by one and is used as identifier for this new EMPLOYMENT entry. `getProfileFromCreateWork()` will return the added profile. The structure of the returned profile has changed. Now it contains super entry WORK and all sub entries of WORK.

1.2.2 Accessing entry content

The next and last part in section Profile is a bit tricky. How can stored content finally be accessed? First it is important to know the structure of an entry. The generic approach makes it hard to access the information without knowing the structure of entries. To access the information just cast to the right type. But you can‘t just type check because the information is from a generic container. Be sure to cast right. In the following example the implementation of entry access will be realized in getters. Getters are added to class CreateWorkEntryImpl. CreateWorkEntryImpl is renamed to WorkProfileImpl and CreateWorkEntry is renamed to WorkProfile.

```
//Interface that contains String constants as identifiers for entries
public interface WorkProfile {
    String WORK = "Work";
    String OCCUPATION = "Occupation";
    String SKILLS = "Skills";
    String EMPLOYMENTS = "Employments";
    String EMPLOYERNAME = "EmployerName";
```

```
    String JOBTITLE = "JobTitle";
    String START = "Start";
    String END = "End";
    String CURRENT = "Current";
    String JOBDESCRIPTION = "JobDescription";

    void setOccupation(String occupation) throws SharkKBException;
    String getOccupation() throws SharkKBException;
    void setSkills(String skills) throws SharkKBException;
    String getSkills() throws SharkKBException;

    void addEmployment(String employerName, String jobTitle,
            int start, int end, boolean current,
            String jobDescription) throws SharkKBException;

    String getEmployerName(String employmentNumber)
                                    throws SharkKBException;
    String getJobTitle(String employmentNumber)
                                    throws SharkKBException;
    int getStartOfEmployment(String employmentNumber)
                                    throws SharkKBException;
    int getEndOfEmployment(String employmentNumber)
                                    throws SharkKBException;
    Boolean getIsEmploymentCurrent(String employmentNumber)
                                    throws SharkKBException;
    String getJobDescription(String employmentNumber)
                                    throws SharkKBException;
}

public class WorkProfileImpl implements WorkProfile {

      //private member variable profile
      private Profile p;

      //The constructor needs an profile as parameter
      //e.g. aliceProfile from class CreateProfile
      WorkProfileImpl(Profile p)  throws SharkKBException {
        this.p = p;
```

```
  createWork();
}

//Create general structure of WORK with no content
private void createWork() throws SharkKBException {
  p.createProfileEntry(WORK);
  p.addSubEntryInEntry(WORK, OCCUPATION, "");
  p.addSubEntryInEntry(WORK, SKILLS, "");
  p.addSubEntryInEntry(WORK, EMPLOYMENTS);
}

public void setOccupation(String occupation) throws SharkKBException {
  p.addSubEntryInEntry(WORK, OCCUPATION, occupation);
}

public String getOccupation() throws SharkKBException {
  Entry<?> entry = p.getProfileEntry(WORK);
  return (String) entry.getEntryFromList(OCCUPATION).getContent();
}

public void setSkills(String skills) throws SharkKBException {
  p.addSubEntryInEntry(WORK, SKILLS, skills);
}

public String getSkills() throws SharkKBException {
  Entry<?> entry = p.getProfileEntry(WORK);
  return (String) entry.getEntryFromList(SKILLS).getContent();
}

//Creates EMPLOYMENT entries and consecutively numbered them
public void addEmployment(String employerName, String jobTitle,
      int start, int end, boolean current,
      String jobDescription) throws SharkKBException {

  //A list of entries is created with parameters as content
  List<Entry<?>> entryList = new ArrayList<Entry<?>>();
  entryList.add(new EntryImpl<String>(EMPLOYERNAME, employerName));
  entryList.add(new EntryImpl<String>(JOBTITLE, jobTitle));
```

```
  entryList.add(new EntryImpl<Integer>(START, start));
  entryList.add(new EntryImpl<Integer>(END, end));
  entryList.add(new EntryImpl<Boolean>(CURRENT, current));
  entryList.add(new EntryImpl<String>(JOBDESCRIPTION, jobDescription));

  //Producing the consecutively numeration
  int count = p.getSubEntry(WORK, EMPLOYMENTS).getEntryList().size();

  //Sets the content "entryList"
  //with a number as name under entry EMPLOYMENTS
  p.createSubEntry(WORK, EMPLOYMENTS,
       Integer.toString(count), entryList);
}
public String getEmployerName(String employmentNumber)
                              throws SharkKBException {
  Entry<?> entry = p.getSubEntry(WORK, employmentNumber);
  List<Entry<?>> entryList = (List<Entry<?>>) entry.getContent();
  return (String) entryList.get(0).getContent();
}

public String getJobTitle(String employmentNumber)
                              throws SharkKBException {
  Entry<?> entry = p.getSubEntry(WORK, employmentNumber);
  List<Entry<?>> entryList = (List<Entry<?>>) entry.getContent();
  return (String) entryList.get(1).getContent();
}

public int getStartOfEmployment(String employmentNumber)
                              throws SharkKBException {
  Entry<?> entry = p.getSubEntry(WORK, employmentNumber);
  List<Entry<?>> entryList = (List<Entry<?>>) entry.getContent();
  return (Integer) entryList.get(2).getContent();
}

public int getEndOfEmployment(String employmentNumber)
                              throws SharkKBException {
  Entry<?> entry = p.getSubEntry(WORK, employmentNumber);
  List<Entry<?>> entryList = (List<Entry<?>>) entry.getContent();
```

```
    return (Integer) entryList.get(3).getContent();
  }

  public Boolean getIsEmploymentCurrent(String employmentNumber)
                                  throws SharkKBException {
    Entry<?> entry = p.getSubEntry(WORK, employmentNumber);
    List<Entry<?>> entryList = (List<Entry<?>>) entry.getContent();
    return (Boolean) entryList.get(4).getContent();
  }

  public String getJobDescription(String employmentNumber)
                                  throws SharkKBException {
    Entry<?> entry = p.getSubEntry(WORK, employmentNumber);
    List<Entry<?>> entryList = (List<Entry<?>>) entry.getContent();
    return (String) entryList.get(5).getContent();
  }
}
```

Now the class WorkProfileImpl is able to store and access occupations, skills and employments. Missing setters and getters for occupation and skills are added too. Heads of new functions are added to interface WorkProfile. Heads of function means the declaration of a function that specifies the function's name and type signature (parameter types, and return type), but omits the function body. How does these getters work? Lets simply explain this with an example: `getEmployerName(String employmentNumber)` . First extracting the right entry. The user of the getter has to specify an employmentNumber. These parameter is like an index for EMPLOYMENTS. Remember that when an EMPLOYMENT is created its indexed by a number. So the first created EMPLOYMENT gets an index of 1 and the second created EMPLOYMENT gets an index of 2 and so on. For example the user wants employer name from the second created employment, so he calls `getEmployerName("2")`. Function `Entry<T> getSubEntry(Entry<T> rootEntry, String subEntryName)` can now search for an entry with "2" as identifier. Variable Entry¡?¿ entry gets the found result. Short explanation of "getSubEntry(...)": these function searches iteratively through a given rootEntry, it searches a specified subEntryName or the name of an entry. If the right entry is extracted, content of entry has to be ex-

tracted and casted. `addEmployment(...)` stores a List of entries as content. To access these content just cast `List<Entry<?>>`. Its possible to make it even shorter: `List<Entry<?>> entryList = (List<Entry<?>>) p.getSubEntry(WORK, employmentNumber).getContent();`. These entry list contains all information about employer name, job title, start of employment and so on. Access the elements of the list with an index. Content of list element one "employerName" is accessed with entryList.get(1).getContent() and content of list element two is accessed with entryList.get(2).getContent(). Last step is returning the casted type of the list element content. `return (String) entryList.get(1).getContent();`. In short:

1. Know the structure and types of entries to access them or cast to the right types
2. Parametrize getters so users can enter what entries they search
3. Use `getSubEntry(...)` to search for entries
4. Access content of entries with .getContent()
5. First extract the entry list then get the right list element
6. Return the correct casted content of the list element

1.3 Summary

Profiles are a good opportunity to store and access a big variety of information about a peer or person. It may be even possible to make profiles about places or historical persons. To manage, create or access profiles just use the profile factory. Customize profiles with the entry functionalities. Store and access different content types or information in customizable entry structures. Contents and information are stored persistently in the SharkKB.