Exploring Role-Based Adaptation

Sebastian Götz, intermediate defense “Großer Beleg” 29.05.08
1. Motivation
2. Key idea [5]
3. Realization
4. Discussion
5. Future Work
1. Motivation

- **Student**
  - printGrades() : void

- **Report**
  - printReport(data: DataRows) : void

- **DBComponent**
  - retrieve(q: DBQuery) : ResultSet

- • already existed
- • is to be improved
1. Motivation

→ Adapter Design Pattern [1] applied twice
→ intertwined code

 Roles annotated as in [2]
1. Motivation

→ expert of 3 domains for printGrades() needed
→ and experts leave

Student

printGrades() : void

Report

printReport(data: DataRows) : void

University Domain

Reporting Domain

Database Domain

DBComponent

retrieve(q: DBQuery) : ResultSet
→ implement adapter roles as first-order programming constructs

- **Student**
  - printGrades() : void

- **Adapter 1&2**

- **Report**
  - printReport(data: **DataRows** ) : void

- **Adaptee 1**

- **Adaptee 2**
  - retrieve(q: **DBQuery**) : **ResultSet**
2. Key idea

Implement adapter roles as firstorder programming constructs

- Student
  - printGrades() : void

- Report
  - printReport(data: DataRows) : void

- DBCOMPONENT
  - retrieve(q: DBQuery) : ResultSet

- Target 1&2

- Adaptee 1

- Adaptee 2

- Adapter 1&2
2. Key idea

- split adapter role

- separation of concerns on the level of implementation
2. Key idea

➔ use ObjectTeams [3]
➔ roles are played by actors in a context [4, p. 66]
2. Key idea

⇒ introduce **intermediate types/methods** to decouple
public team class StudentAdapter {
    private ReportOut report;
    private DBOut db;
    private Map<String, Double> grades;

    public StudentAdapter(Report report, DBComponent db) {
        this.report = new ReportOut(report);
        this.db = new DBOut(db);
    }

    public class StudentIn playedBy Student {
        printGrades <- replace printGrades;
        public callin printGrades() {
            grades = db.getGrades();
            report.print(grades);
        }
    }

    //...
public class DBOut playedBy DBComponent {
  Map<String,Double> getGrades()
  -> ResultSet retrieve(DBQuery q)
  with { new DBQuery(„SELECT * FROM Grades“) -> q,
    result <- transformToMap(result) }

  Map<Subject,Grade> transformToMap(ResultSet rs) {
    Map<String,Double> ret = new HashMap();
    while (rs.next() != null) {
      ret.put(rs.getString(1), rs.getDouble(2));
    }
    return ret;
  }
}
public class ReportOut {
    public void print(Map<String,Double> map) {
        printReport(transformToDataRows(map))
    }

    void printReport(DataRows rows) {
        for (String subject : map.keySet()) {
            rows.addString(subject + ' - ' + map.get(subject));
        }
    }

    private DataRows transformToDataRows(Map<String,Double> map) {
        DataRows ret = new DataRows();
        for (String subject : map.keySet()) {
            ret.addString(subject + ' - ' + map.get(subject));
        }
        return ret;
    }
}
4. Discussion

- **separate roles for separate concerns**
  - Hence, no more tangling code
  - Easier extensible (just add a new role)
  - Intermediate types decouple domains
  - **lower maintenance costs**

- **roles enable unanticipated adaptation**
5. Future Work

- evolution of integrated components → role composition

- Upgrade ReportOut invasively?
- Generate additional ReportOut (from change information)
- and compose with old ReportOut!


Thank you very much for your attention.

Any Questions?