Hands-on introduction to Object Teams

Stephan Herrmann, GK Software AG
Olaf Otto, Unic AG

March 24th, 2011

EclipseCon 2011, Santa Clara
Hands-on Introduction to Object Teams

- With good connectivity:
  - follow links from the EclipseCon program page

- Else unpack these archives from USB to your hard disk
  - OTEclipseConTutorial.zip
    - creates a directory OTEclipseConTutorial/
    - the Eclipse SDK for your platform
  - done with the USB stick

- Within Eclipse (new empty workspace!) install OTDT:
  - local repository: OTEclipseConTutorial/otdt-updateSite
  - select everything (two features) & install

- Open tutorial data
  - within OTDT import existing project
    - OTEclipseConTutorial/StarterKit.zip
  - slides are in OTEclipseConTutorial/Slides.pdf
Speaker Introduction (1/2)

**Speaker Information**

Stephan Herrmann
- Dr.
- Software Architect
- Project Lead
- JDT/Core Committer
- Xtext

```
stephan:Person
name: "Herrmann"
firstName: "Stephan"
title: "Dr."
```

**Technical Details**

- NPE
Model Review

stephan:Person

- name: “Herrmann”
- firstName: “Stephan”
- title: “Dr.”

SoftwareArchitect

JDT/Core

Committer

ProjectLead

«instanceof»

«instanceof»

«instanceof»

GK SOFTWARE

eclipse.org/objectteams

EclipseCon 2011, Santa Clara | © 2011 by Stephan Herrmann and Olaf Otto; made available under the EPL v1.0
Not Classes – Roles!

:SoftwareArchitect

:ProjectLead

:Committer

stephan:Person

name: “Herrmann”
firstName: “Stephan”
title: “Dr.”
Classes vs. Roles

- Roles are instances
  - modeled by classes

- «roleof» is an instance relation
  - modeled by a playedBy relation

- Roles are dynamic
  - add at runtime
  - remove at runtime

- Roles are independent
  - object may have several roles ...
  - … even of the same type
    („I am 2 committers“)
Speaker Introduction (2/2)

olaf:Person

name: “Otto”
firstName: “Olaf”
title: “Dipl. Inf.”

:SoftwareArchitect

:CMS expert

:Committer
- **OT/J**
  - Java += roles, teams, bindings

- **Object Teams Development Tooling**
  - Java Compiler += OT/J constructs
  - JDT for OT/J (code assist, ui, launch …)

- **OT/Equinox**
  - Equinox += aspect bindings

- **Eclipse Object Teams Project**
  - Indigo train += OTDT
  - Planning to graduate for Indigo
Why?

- modularity
- encapsulation
- maintenance
- refactoring
- re-use
- design intent
- scalable structures
- generalization
- evolution
- composition
- specialization
- variants
- product line
- context
- prototyping
- views
- adapt
- adapt Equinox Java collaboration
- Object
- Object teams
- SEPARATION
- OF
- composition
Lesson 1

Use Case Modules

Transfer Money
Lesson 1: Use Case Modules

- **Separation of layers**
  - domain layer: pretty dumb objects
  - use case layer: behavior implemented as roles
    for maximum re-use: make border permeable

- **Wiring of layers**
  - bind classes
  - bind instances
  - bind methods

- **Use case API**
  - entire use case is one class
  - instantiate and invoke

---

**Motto:**

Modules are boring
Connections are where the music plays
Split: behavior object → domain object

- **role** → base

Access to domain objects

- only via role (“gateway”)
- access to base methods via “callout” method binding
Use Case Modules: Concepts (2/2)

- **Use case module**
  - **team** class & team instance
  - container for roles

- **Instantiate**
  - teams
  - roles inside a team instance
    - role instance needs a base instance
    - may invoke **base** constructor
Use case modules: Syntax

```java
public class Person {
    private String name;
    public Person(String name) {
        this.name = name;
    }
    public String getName() {
        return this.name;
    }
}

public team class Company {
    protected class Employee playedBy Person {
        private String officePhoneNo;
        String getName() -> String getName();
    }
    ...
}
```
### Use case modules: Syntax

```java
public class Person {
    private String name;
    public Person(String name) {
        this.name = name;
    }
    public String getName() {
        return this.name;
    }
}
```

```java
public team class Company {
    protected class Employee playedBy Person {
        private String officePhoneNo;
        String getName() -> String getName();
    }
    ...
}
```
Use case modules: Syntax

```java
public class Person {
    private String name;
    public Person(String name) {
        this.name = name;
    }
    public String getName() {
        return this.name;
    }
}
```

```java
public class Company {
    protected class Employee {
        private String officePhoneNo;
        String getName() {
            return this.officePhoneNo;
        }
    }
    ...
}
```
Code-along Lesson 1.1

- Create team “Company”
- Create role “Employee”
- Create callout to “getName()”
Obtaining and wiring instances

- Explicit role creation
- Implicit role creation
Obtaining and wiring instances

- **Explicit role creation**
  - requires a base instance
    - e.g. by passing an existing base as argument

- **Implicit role creation**
Obtaining and wiring instances

- **Explicit role creation**
- **Implicit role creation**

- already have a **team instance** and a **base instance**
- team and outside use this base as a shared handle
  - do you know “Olaf”?
    - “Ah, that funny guy!” (Person)
    - “Oh, he's our best software architect!” (Employee)
- team translates the base to an appropriate role: “**lifting**”
- translation may **create** a role (on-demand)
- when to lift?
  - rule-of-thumb: lifting affects all **data flows ...**
    - entering a team instance
    - involving a base instance
    - for which a bound role class exists
Use Case API

- Team class as a façade to hidden roles
  - new syntax “declared lifting”

    ```java
    void hire(Person as Employee emp, ...)
    ```

  - partial information sharing
    - outside (client): pass base instance, cannot see roles
    - inside (team): treat as a role, should mention base only after `playedBy`

  - inverse: lowering
    - team trying to pass a role to the outside
    - the outside will only see the base instance
Create API methods in “Company”

- Method “hire(Person)"
- Method “getBusinessCard(Person)”
Exercise 1.1: Money transfer between accounts

- Given a pretty dumb base class `world.Account`
- Write a use case module (team) for the transfer
  - distinct `role classes` for the `participating objects`
    - access base members using `callout`
  - implement this use case in a non-static team method:
    
    "if amount can be withdrawn from the source, let the sink accept this amount"
  - handle `insufficient funds` inside the source account
  - optional: add simple `checkPoint/rollBack capability` to source account

- Implement API as required by `TransferTest`:
  - provide methods for invoking the transfer from the outside
    - clients cannot see role types, but still invoke role behavior
    - client will provide `Accounts` as arguments

- Hint: to make roles printable declare (overriding callout):
  
  `toString => toString;`
Exercise 1.2: Generalization

Support loading a pre-paid phone, too

- generalize: extract a role interface `IMoneySink`
  - use the interface where appropriate
- create a new role class bound to `PrepaidPhone`
- uncomment statements for testing the new case
Exercise 1: Summary

**Roles** are context specific views
- bound to base class using `playedBy`
- base properties are exposed using “callout”
- more properties can be added as needed

**Teams** define context for interacting roles
- team = container & façade
- data flows into/out off team: “lifting” / “lowering”

**Post-hoc generalization**
- roles can abstract about unrelated base classes
  - unbound super role (interface)
  - more specific roles bind to individual base classes
Lesson 2: Adapting Existing Behavior (1/2)

A role may intercept calls to its base
- “callin” method binding
- inverse to callout
- three flavors: before, after, replace

No pre-planning
- neither base object nor its caller need to know
public class Person {
    private String phoneNo;
    public String getPhoneNo() { return this.phoneNo; }
}

public team class Company {
    protected class Employee playedBy Person {
        private String officePhoneNo;
        String getPhoneNo() String getPhoneNo();
        String getPhoneNo() { return officePhoneNo; }
    }
}
Create callin method & binding in “Company”
  Person should answer his/her **officePhoneNo**.
Lesson 2: Adapting Existing Behavior (2/2)

Enablement at different levels

- main switch: **team activation**
  - methods activate / deactivate of class `org.objectteams.Team`
  - block construct “within”
  - global / per thread
- fine tuning: **guard predicates**
  - per role class
  - per callin binding
Try different ways to activate “Company”
  - methods activate/deactivate
  - within
Exercise 2: Transaction

Transfer and Talk are concurrently modifying Phone

- Implement a transaction that synchronizes talk().
- The transaction synchronizes access to its participants while it is active.
  The participants are only modifiable by the current thread. Other threads are blocked until the transaction is deactivated.

Hints

- Use the provided TransferTest for TDD
- A java.util.concurrent.ReentrantLock() is a great tool for thread-exclusive locking
Exercise 2: Cheat sheet

- Calling bindings
  - roleMethod <- [before | after | replace] baseMethod

- Team Activation
  - activate(): Activates a Team for the current thread
  - activate(Thread): Activates a team for the given thread, activate(Team.ALL_THREADS) activates it for all.

- Advanced tips:
  - A guard predicate of the form protected class MyRole … base when(hasRole(base)) {…} can be used to prevent automatic role creation.
  - A team method of the form (BaseType as RoleType identifier) creates a role instance in the team.
  - within(Team) { … } activates a team for its scope and the current thread.
Exercise 2: Summary

Achievements:

- Factored out the pervasive synchronization aspect
  - No changes in base and transfer
  - Entire synchronization context reified in team
- Synchronization on demand

Used techniques:

- Callin bindings
- Guard predicates
- Team activation
The dilemma of SW evolution

For implementing & integrating a new feature
- need to modify many existing classes & their structure

For comprehensibility, maintainability
- need to keep changes well localized
Solution: add another Dimension

- Zoom out off the base plane
- Define suitable structure using teams & roles
- Create connections using playedBy, callout & callin
Lesson 3: Superimpose Structure

Find employees living in the same city for ride sharing

- needs relation City →* Person
  - this relation is missing from the model
  - this relation *may* be unwanted in the domain layer
- consider possible solution Relation Manager
  - may be considered an anti-pattern
  - want information right in the objects operating on it
- relation is a must, but we cannot pay for it
Find employees living in the same city for ride sharing

- **Forget about** City →* Person
  - instead add a new role HomeTown playedBy City and ...
- **Add missing relation** HomeTown →* Employee
  - add collection commuters in role HomeTown
  - maintain collection during **hiring** of people
    - using inverse relation livesIn: Person → City

**Find ride sharing**

- iterate all known HomeTowns
  - using API ITeam.getAllRoles(Class<?> roleClass)
- iterate all registered commuters
Lesson 3: Summary

- Not just classes: structure, too.
  - views on existing **relations**: “callout”
    - `getLivesIn()`: Employee → HomeTown, view on Person → City
  - add new **relation**
    - `commuters`: HomeTown → Employee, derived from `getLivesIn()`

- Team is a view of a base model
  - role → base class & object
  - callout → base method / field
  - inheritance
  - relations
  - All:
    - selectively expose existing
    - adjust
    - and add more
Bonus Exercise

- Implement the following demo-mode for the JDT
  - When creating a Java project let the user select:
    - Project is for demo purpose only
  - When creating a class in a Demo project
    - Insert class name as “Foo1”, “Foo2” ...

Hints into the JDT/UI and JDT/Core (incomplete)

- org.eclipse.jdt.ui.wizards.NewJavaProjectWizardPageOne.NameGroup
- org.eclipse.jdt.core.IJavaProject
- org.eclipse.jdt.ui.wizards.NewTypeWizardPage

Configuration hints

cf. [http://wiki.eclipse.org/Object_Teams_Quick-Start#OT.2FEquinox_Hello_World_Example](http://wiki.eclipse.org/Object_Teams_Quick-Start#OT.2FEquinox_Hello_World_Example)

- new Object Teams Plug-in Project
- Dependencies: org.eclipse.jdt.ui & org.eclipse.jdt.core
- Extension: org.eclipse.objectteams.otequinox.aspectBinding
  - One extension for each affected base plug-in
  - Don't forget “activation”
Summary I

Questions?

«playedBy»

Off  On
More questions?  
mailto:stephan@cs.tu-berlin.de  
mailto:olaf.otto@unic.com  