Modularity

Collaboration Modules

Team classes – modules larger than classes

Teams combine desirable properties from **classes** (OOP) and **packages** (source code organization). As classes teams are instantiable, have attributes and methods and support inheritance. As packages teams support nesting which is optionally realized by a hierarchy of directories and files.

Role classes – relative entities

Each role instance *lives in* exactly one enclosing team instance, thus participating in one collaboration. A role class can be bound to a base class using a **«playedBy»** relationship (see →**Adaptability**)

Team Inheritance

Applying inheritance to compound modules

Inheritance cannot only be applied to atomic classes but also to a whole team, even if it contains nested teams (see →**Scalability**).

Implicit role inheritance

A role with the same name as an inherited role implicitly overrides the existing role and inherits its features. Thus a team can be used as a framework, where all features (at any level of nesting) are hotspots which can be overridden in a sub-team.

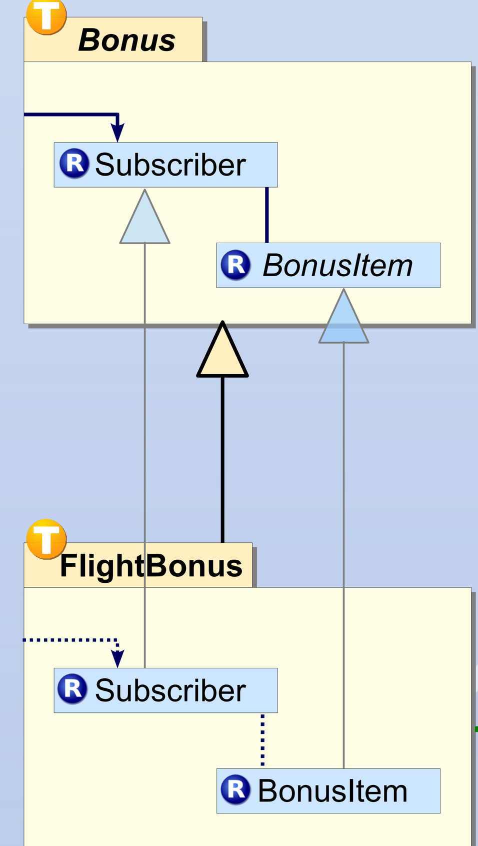
Consistent refinement of relations between roles

A relation defined between roles of a super-team is implicitly adjusted to consistently connect roles of the sub-team. So within the `FlightBonus` team, the inherited association `Subscriber ↔ BonusItem` automatically refers to `FlightBonus.Subscriber` and `FlightBonus.BonusItem`, making it impossible to mix roles from different teams.

Role instantiation

`new BonusItem()`

Creating a role instance determines the role class to use from the enclosing team instance. Thus, the tedious Factory design pattern is no longer needed.



FlightBooking

Passenger

Flight

Segment

«adapts»

Motivation

Limitations of Object-Oriented Programming

Scale

While object-oriented programming works very well in the small, creating a system from 1000s of classes does not sufficiently support a good modular structure. Classes need to be grouped to scalable modules in order to reduce the number of possible connections between classes.

Complexity

Any single dimension of decomposition does not support to explicitly capture several crosscutting concerns. As a result an object-oriented implementation of **crosscutting concerns** yields concerns that are **scattered** over many classes, while in each of these classes several concerns occur **tangled** with each other.

Long-term Maintenance and Evolution

For any given point in time the above problems *could* be tolerated, but if a software system is supposed to be used over a long period of time, in which it must be maintained and should evolve, superior modularity is required to prevent sky-rocketing development costs.

Sustainable Software-Engineering requires

... significant advances in ...

Modularity

All concerns should be encapsulated in modules with well-defined boundaries.

+ Scalability

Techniques for composing a system from components must be applicable at any scale.

+ Adaptability

Existing components must support various adaptations, anticipated *and* unanticipated.

= Reusability

Only components supporting modularity, scalability *and* adaptability can be efficiently re-used in other systems and in the future.

... even in situations where these seem to conflict with each other.

Object Teams provides

Collaboration Modules

Team classes, grouping sets of collaborating roles, provide clean modularity and encapsulation even for crosscutting concerns.

Role Binding

The playedBy relationship enables a role to transparently decorate and/or adapt its bound base.

Team Inheritance

By applying inheritance to compound modules, reusability is significantly improved.

Compositionality

The relationships *role containment*, *role binding* and *team inheritance* can be composed freely in order to create larger structures.

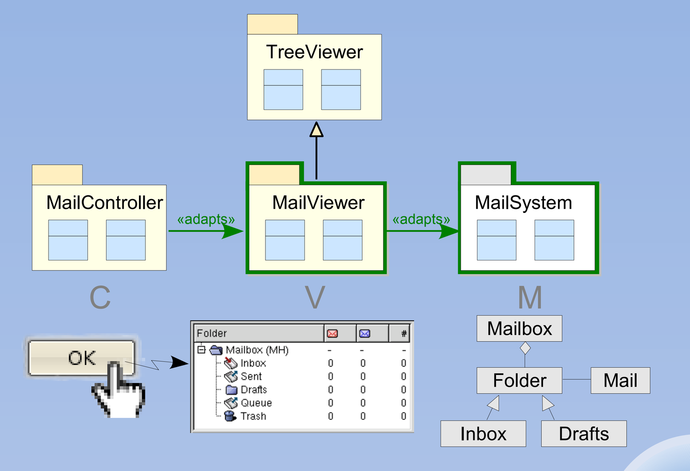
Tools

A wide spectrum of well-integrated tools supports high productivity for developing quality software in ObjectTeams/Java.

Fields of Application

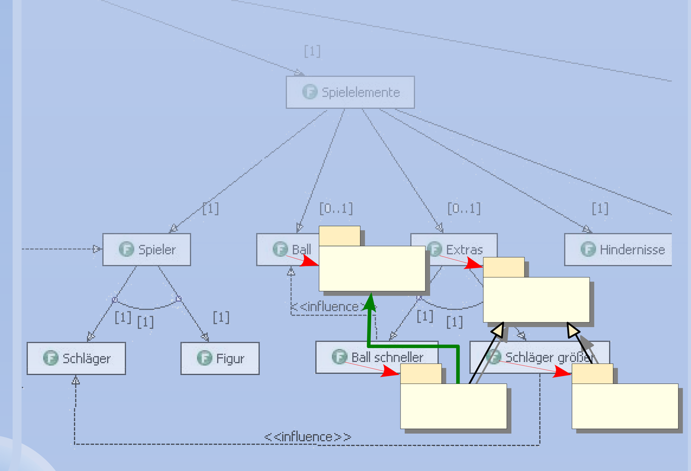
Graphical User Interfaces

With Object Teams, the well-known **M**odel-**V**iew-**C**ontroller architectural pattern can be implemented in unprecedented clarity and with optimal re-use.



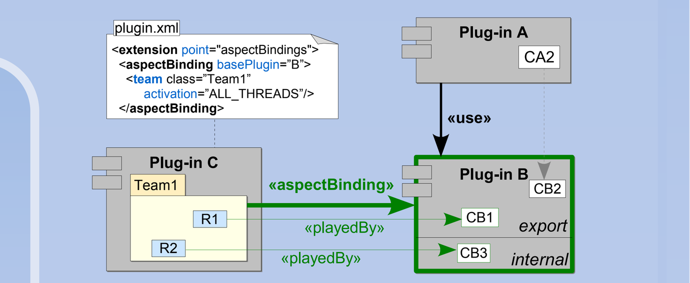
Product Lines

Feature models, which are the basis for product line development, can seamlessly be mapped to Object Teams structures, thus improving traceability and maintainability.



Components

Component platforms like OSGi offer support for larger modules with deployment, dependency and lifecycle management.
To combine the strengths of Object Teams and OSGi, **OT/Equinox** integrates OT/J with Equinox, the Eclipse implementation of the OSGi standard.



OT/Equinox is used in the OTDT, where **aspect plug-ins** adapt existing components from the JDT. Another application is the integration and customization of GMF-based graphical editors.

Embedded Systems

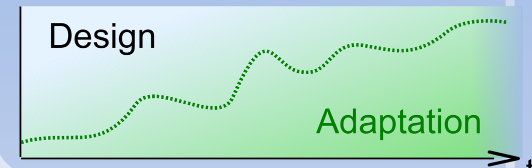
In order to support OT/J applications to run on small devices, an optimized virtual machine is being developed (see →**Tools/Runtime**):

- Implement **aspect dispatch** in the VM rather than weaving extra dispatch code.
- Provide **team activation** as a VM service rather than maintaining administrative data via generated code.
- Avoid byte-code duplication as currently needed to realize **team inheritance** on a standard JVM.

Once small devices can execute OT/J programs, the full benefits of **product line** development as well as **components** (OSGi) can be leveraged in this field, too.

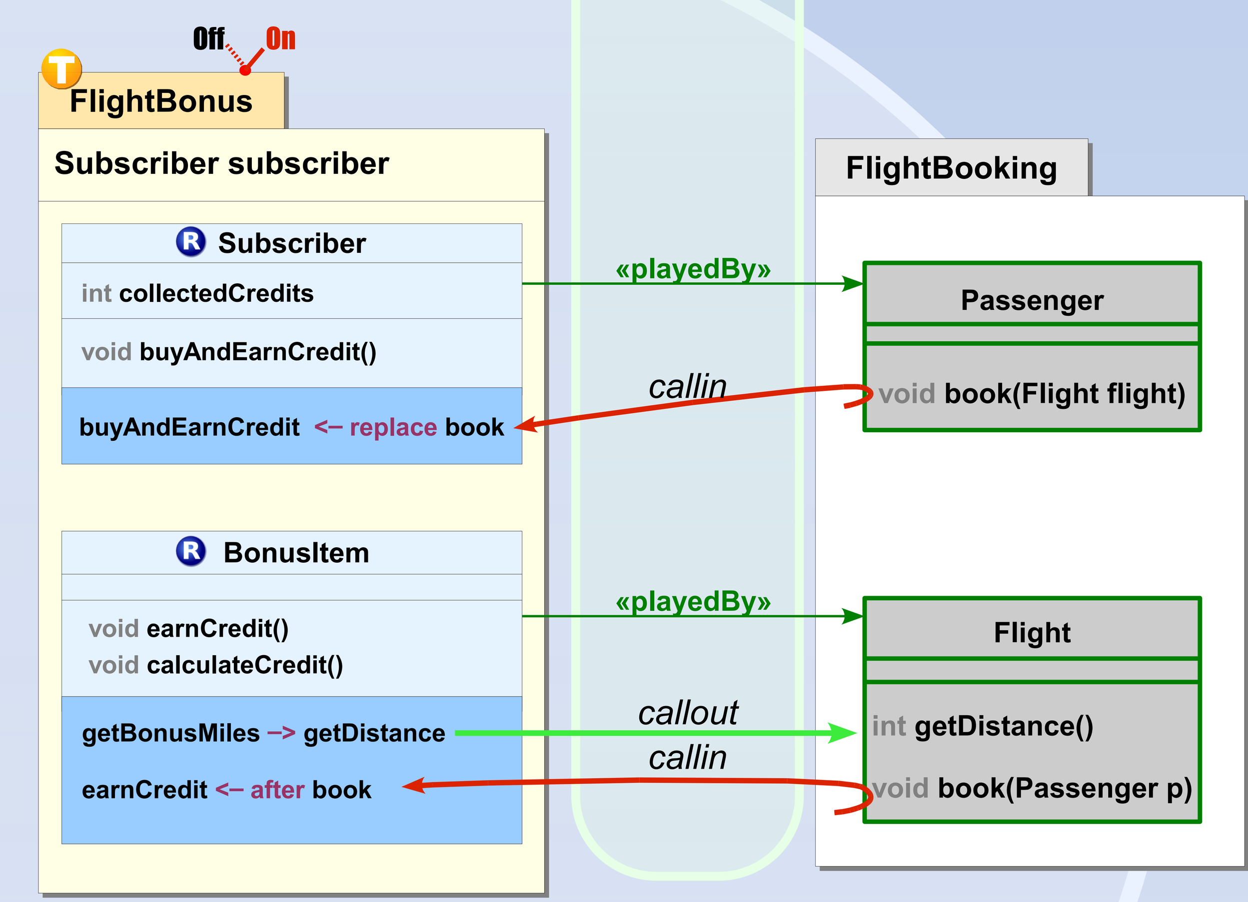
Evolution

As software matures over time, efforts are shifted from design towards adaptation. Both areas are covered by Object Teams.



ObjectTeams/Java
extends the Java programming language
for role-based and aspect-oriented software development
to support the requirements for
sustainable software engineering

Adaptability



Role Binding

Aspect activation

Off / On

Dynamic aspect activation
- per team instance
- per registered base object (with guards)
- per thread or globally

Method bindings

-> Forwarding (***callout***); making base method/field accessible via the role, too.

<- Method call interception (***callin***); behavioral adaptation as used in Aspect-Oriented Programming with *before, after* or *replace*-binding

Dynamic filtering

Guard predicates allow dynamic filtering of callin bindings
- per method
- per role
- per team

Flight-Bonus Example

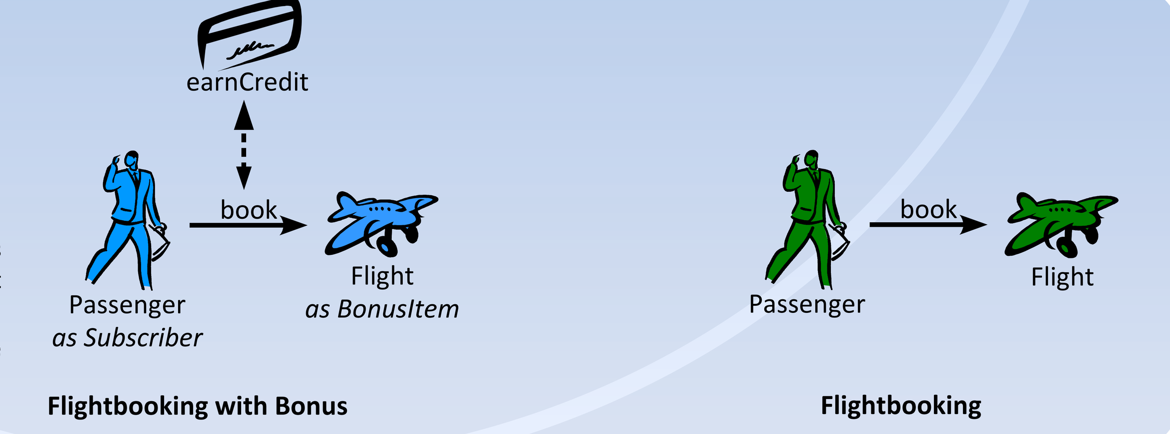
The example shows two requirements encapsulated in two completely independent modules:

`FlightBooking` – a passenger can book flights.

`Bonus` – a subscriber may collect credit points.

`FlightBonus` is realized as a connector team, binding the roles `Subscriber` and `BonusItem` inherited from the `Bonus` context to classes of the FlightBooking system.

Adaptation takes place in the way that a passenger may play the role of a subscriber and his flight plays the role of a bonus-item, simply by using air miles to calculate the creditpoints.



Scalability

Nesting
Team
Stacking
Role
Base
Layering

Compositionality

Programming language features should be orthogonal in a way that allows for flexible use and combination to achieve scalable designs.
To support this compositionality, in Object Teams a class can have up to 3 natures simultaneously: *Team*, *Role*, and *Base*. The three architectural styles shown below capture the most common usage patterns demonstrating orthogonality and scalability in Object Teams:

Nesting

Teams can be nested recursively, allowing teams to act as roles (or: roles being implemented as teams). Thus, nested teams can adapt base classes *and* contain collaborations at the same time.
This structure can help in building complex collaborations and containment hierarchies.

Stacking

Teams are regular classes and as such can certainly be played by roles. With stacking you are able to build collaborations over a set of teams and adapt team-level behaviour. This can be used to coordinate teams and collaborations that are otherwise not related.

Layering

Roles can be played by roles as well, enabling a layered architecture.
Due to the strong encapsulation properties of Object Teams, a team reference is needed in order to refer to roles of another team. Layering supports incremental extensions of collaborations and decorator-style adaptations.

Tools

Language Definition (OTJLD)
Modeling (OTML)
Components (OT/E)
Language (OT/J)
IDE (OTDT)
Compiler
Engineering Testing
Runtime (OTRE)

The Object Teams Ecosystem

Code contributions by
- Fraunhofer FIRST
- Students (diploma theses)

Development
- Automated building/testing
- Trac issue tracker & wiki

Community
- Mailinglist otj-users

Application in
- Case studies (GEBIT Solutions)
- Implementation of the →**OTDT**
- Classes, diploma theses ...

Language Definition (OTJLD)

This document defines the concepts of OT/J and its syntax and semantics. It is published in three formats (web, print, otdt).

Compiler

The ObjectTeams/Java compiler extends the regular Java compiler to reflect the additional features of OT/J. Special instructions for the weaving process are stored in byte-code attributes of compiled team and role classes.

base code
.java
OT/J code
.java
Java Compiler
OT/J Compiler
.class
.class
extended byte-code attributes
.class
external classes

OT/J constructs can adapt elements of plain Java and OT/J implementations, either in source or byte-code.

Runtime (OTRE)

The ObjectTeams/Java Runtime Environment is responsible for aspect weaving. It transforms the byte-code at load-time to weave aspect dispatch code into base classes. This allows for adaptation of classes even if no source code is available.

Load-time weaver
JMangler/JPLIS
.class
woven classes
JVM (standard)

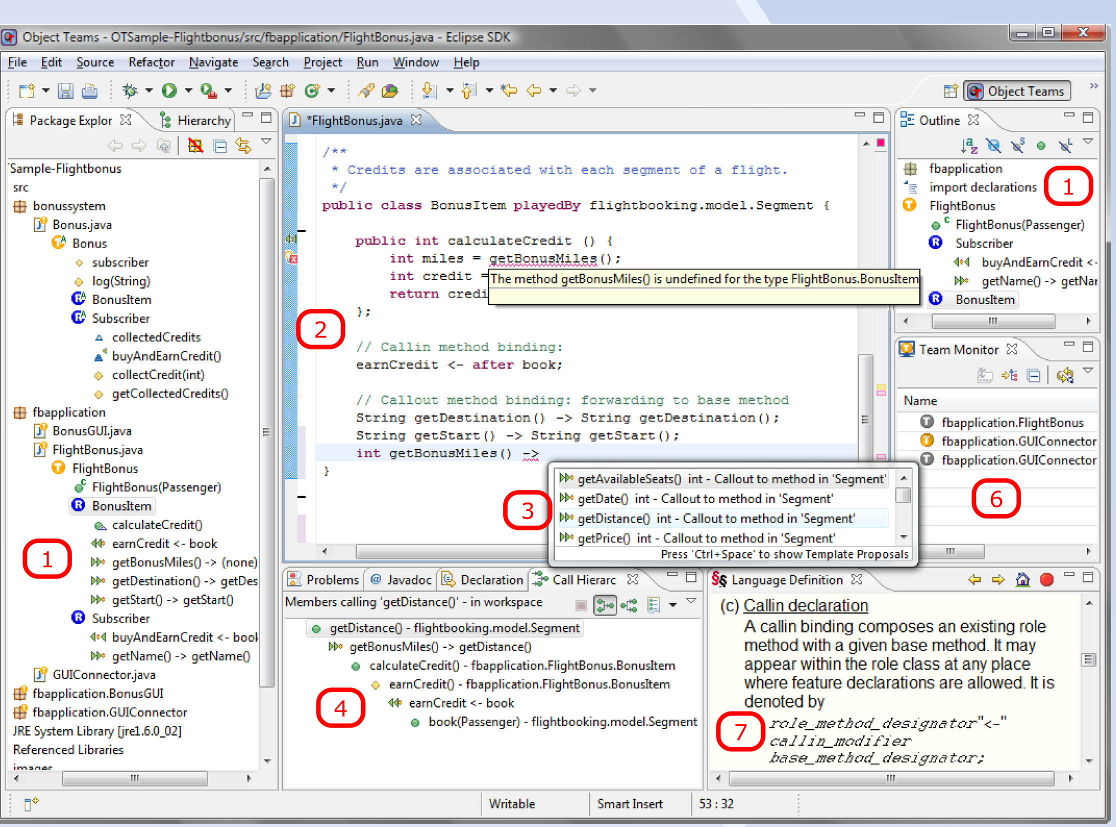
The OTRE is currently implemented based on *JMangler*. An experimental version alternatively uses the *JPLIS* API introduced by Java 5.

In ongoing research, aspect-features are being integrated directly into a JVM to enable more efficient execution, e.g. on mobile devices.

The Object Teams Development Tooling (OTDT)

- supports development of OT/J programs by a rich set of →**Features**
- provides and extends the convenience of the well-known **Eclipse JDT**
- adapts existing Eclipse plug-ins using **OT/Equinox** (→**Components**)
- is developed since 2003 and freely available under **E**clipse **P**ublic **L**icense
- is continuously tested by two comprehensive test suites (white & black box)

BUILT ON eclipse



OTDT
1 2 3 4 5 6 7 8
OT Modeler
OT/E
UML2 Tools
JDT
1 2 3 4
8
PDE
GMF
EMF
GEF
Eclipse platform
Equinox

Features

1 Navigation
2 Editor
3 Code Assist + Quick Fix
4 Call Hierarchy
5 Binding Editor
6 Team Monitor
7 Online Language Definition
8 Aspect Plug-in support
+ Debugger, Refactoring, ...

www.objectteams.org

Contact: Stephan Herrmann (project lead)

Christine Hundt (OTRE, embedded systems)

Marco Mosconi (OT Modeler, components)