EASSy 2013 (Sep. 11, 2013)

Role-based Adaptive Modeling Framework "Epsilon" and a Case Study

Tetsuo Tamai (Hosei University)
collaborated with
Supasit Monpratarnchai (Fujitsu)

Self* and Epsilon Model

- Self-* systems
  - Self-organizing, -managed, -healing, -repairing, -monitoring, -reconfiguring, …
  - Autonomic, adaptive, evolving, …
- Role-based model Epsilon
  - Its motivation was adaptation to dynamic environment changes.

Objects & Environments

- Objects reside in an environment or in multiple environments at a time.
- Objects change their behavior according to environments
  - e.g. day/night, weekdays/weekend
- Objects evolve to adapt to environment changes.

Role Model: Epsilon

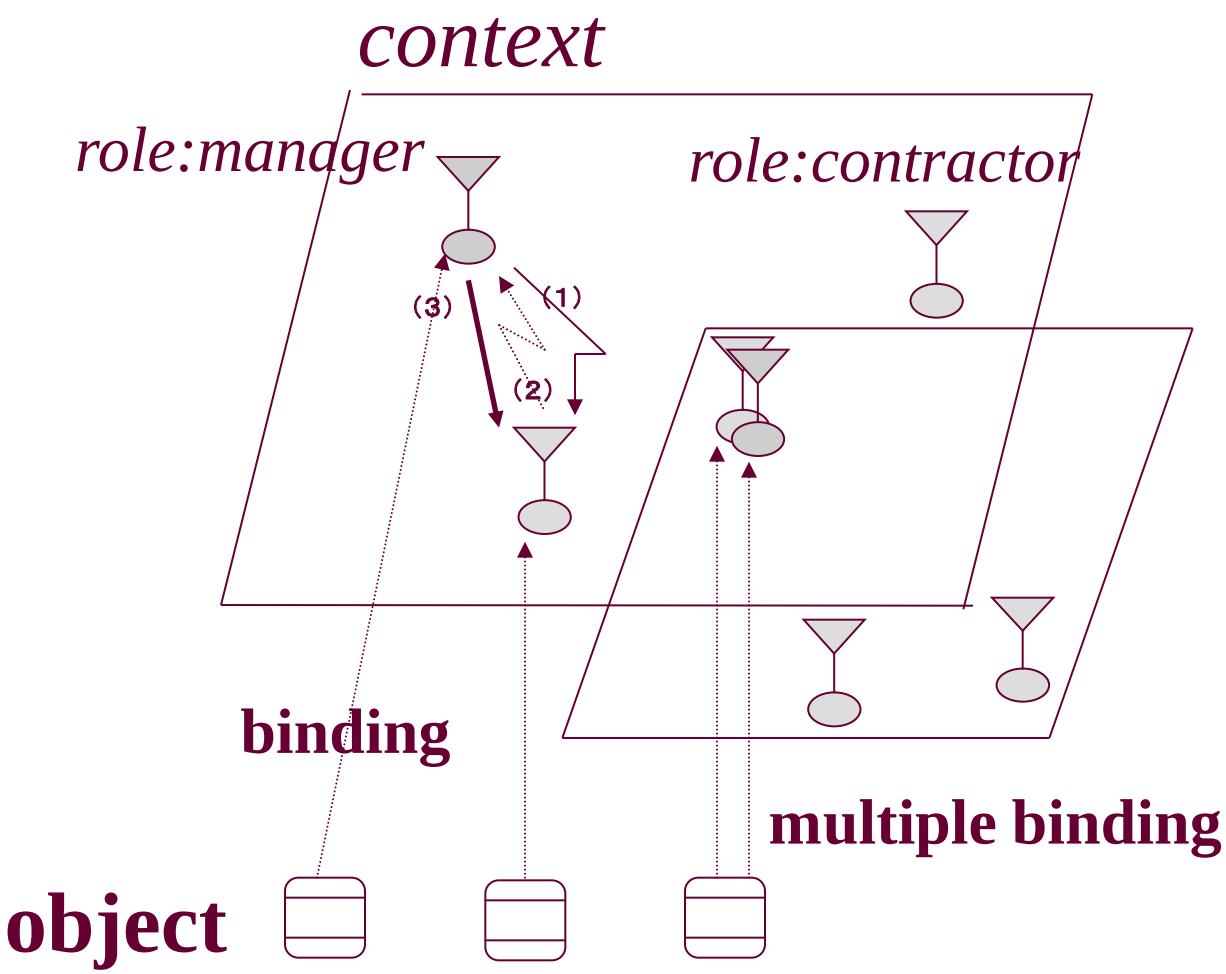
- Design goals:
  - support adaptive evolution
  - enable separation of concerns
  - advance reuse

(Tamai,T. et al., "An Adaptive Object Model with Dynamic Role Binding," ICSE'05)

Features of Epsilon

- Contexts encapsulate collaboration fields enclosing a set of roles.
- Objects freely enter a context by binding to roles and leave a context by unbinding the ties to roles.
- Objects can belong to multiple contexts at a time.
- Contexts (with roles) are independent reuse components to be deployed separately from objects.

Context, Roles and Binding



Programming Language EpsilonJ

- A language based on Epsilon model
- Constructs for defining a context enclosing roles to describe and encapsulate collaboration
- Dynamic binding and unbinding mechanism of an object and a role

Declaration of Context and Roles

```
context Company {
  role Employer {
    void pay() {
      Employee.getPaid();}}
  role Employee {
    int save, salary;
    Employee(int salary) {
      this.salary = salary;}
    void getPaid() {
      save += salary;}}}
```

- Context and roles are like class and inner classes but roles are more concrete and couplings among them are stronger.
  (Context is composition rather than aggregation.)
- Roles can see only each other and methods/fields of its context.

Binding Objects with Role

```
class Person {
  string name;}
Person tanaka=new Person();
Person hamada=new Person();
Company todai=new Company();
todai.Employer.newBind(hamada);
todai.Employee.newBind(tanaka);
```

Objects Acquire New Functions through Binding

```
((todai.Employer) hamada).pay();
((todai.Employee) tanaka).getPaid();
```

- Objects can access to functions of contexts (collaboration fields) only through binding to roles.

Role may have Multiple Instances

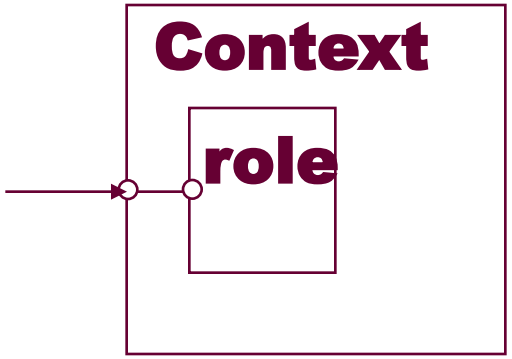
```
Person suzuki=new Person();
todai.Employee.newBind(suzuki);
```

- Binding is between an object and a role instance.
- Both ways of handling those roles as a collection and by instance are provided.

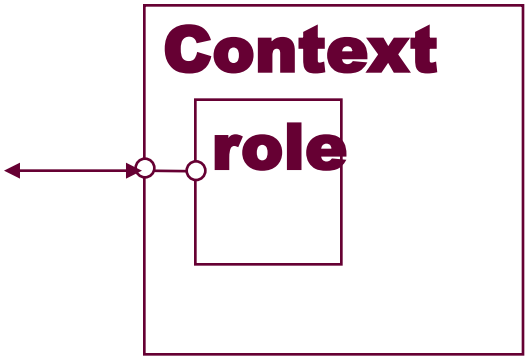
Method Import/Export by Binding

- Binding an object to a role affects states and behavior of the object and the role (interaction between the object and the role).
- Interface for binding can be defined and used in binding.

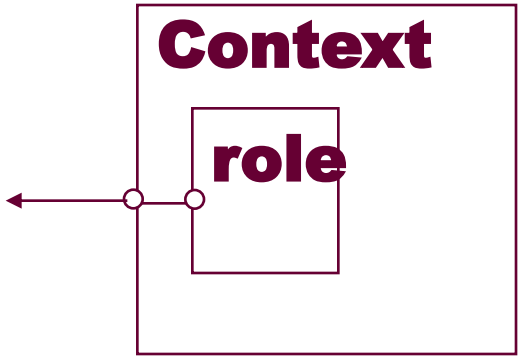
import



import & export (overriding and call of super)



export (overriding)



Examples

- Integrated System
- Mediator Pattern
- Observer Pattern
- Visitor Pattern
- Kendall's example
- Rental shop
- Dining philosophers

Implementation

- EpsilonJ translator http://www.graco.c.u-tokyo.ac.jp/~supasit/epsilonj/
  - Translation from EpailonJ to Java source code
- Model Transformation Framework http://www.graco.c.u-tokyo.ac.jp/~supasit/transformation/
  - Transformation from i* model to EpsilonJ/Java program

Developed by Supasit Monpratarnchai

Epsilon Model Transformation Framework

- RE model: i* SD & SR
  - Actor, Agent, Role, Position
- Representation in UML extension for Epsilon
  - stereo-types: context, role
- Model transformation by ATL
  - → EpsilonJ/Java programs

Requirements Model in i*

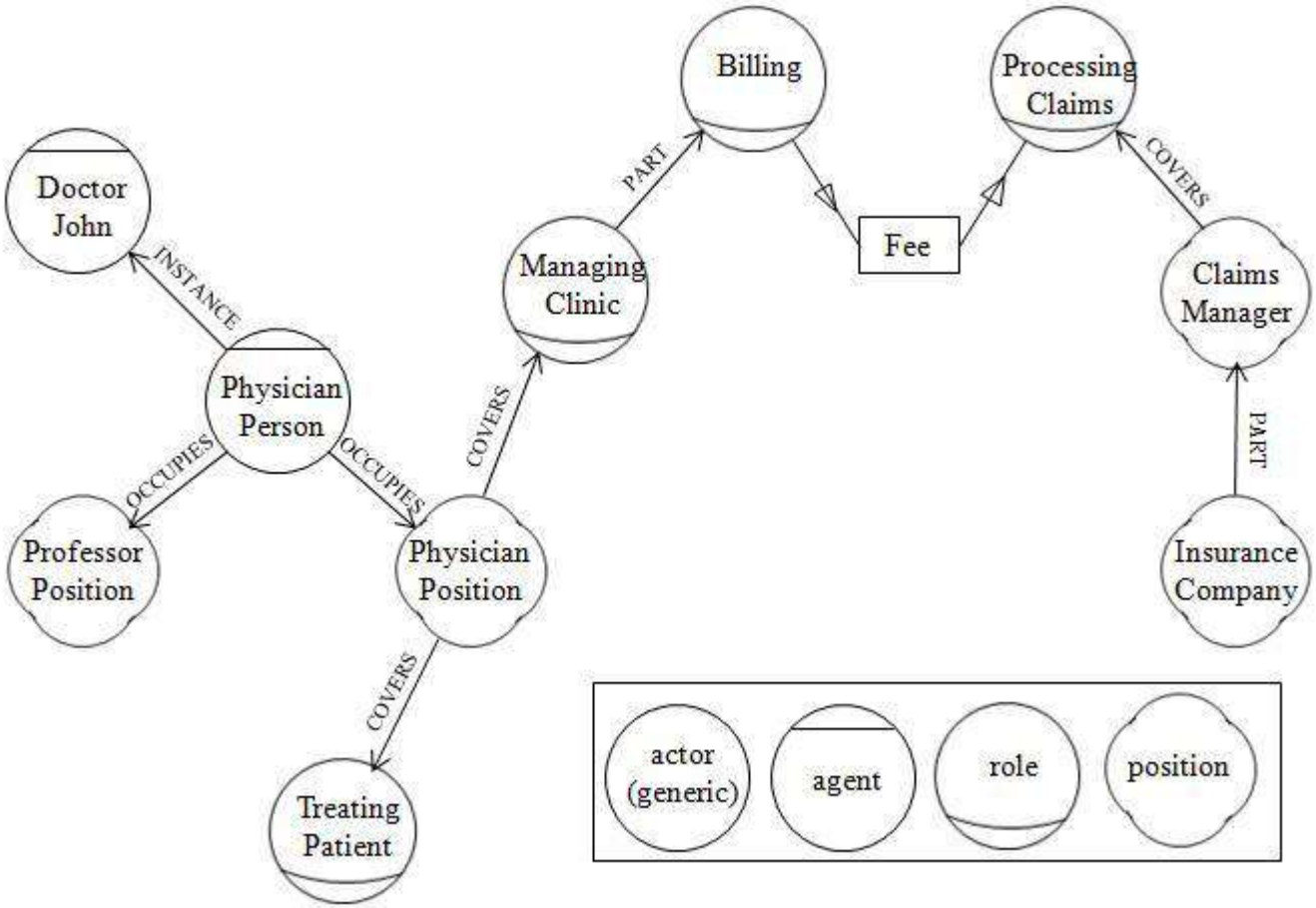
↓

Epsilon Model in UML

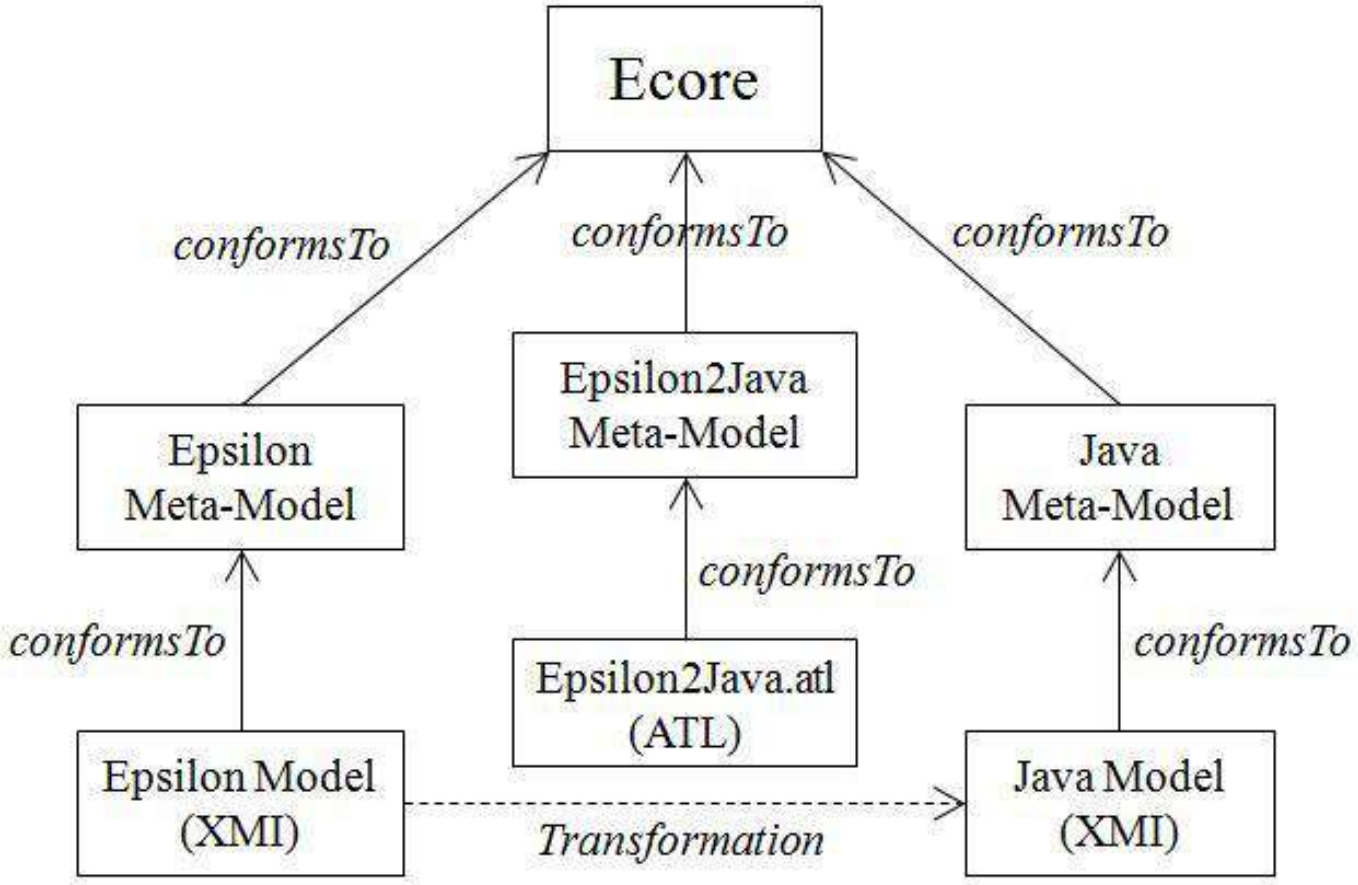
↓ **ATL**

Implementation in EpsilonJ/Java

Agents, Roles and Positions



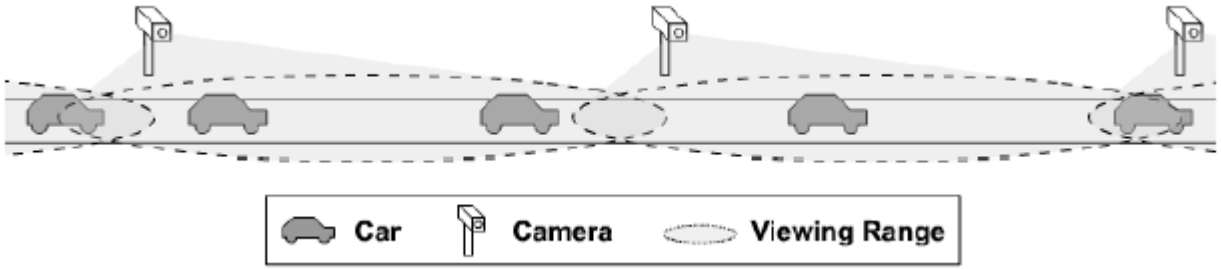
Model Transformation



Example of Self-adaptive System

- Traffic jam monitoring system
- R. Haesevoets et al., "Managing Agent Interactions with Context-Driven Dynamic Organizations," *LNAI 5049*, pp.166–186, 2008.

Traffic Jam Monitoring

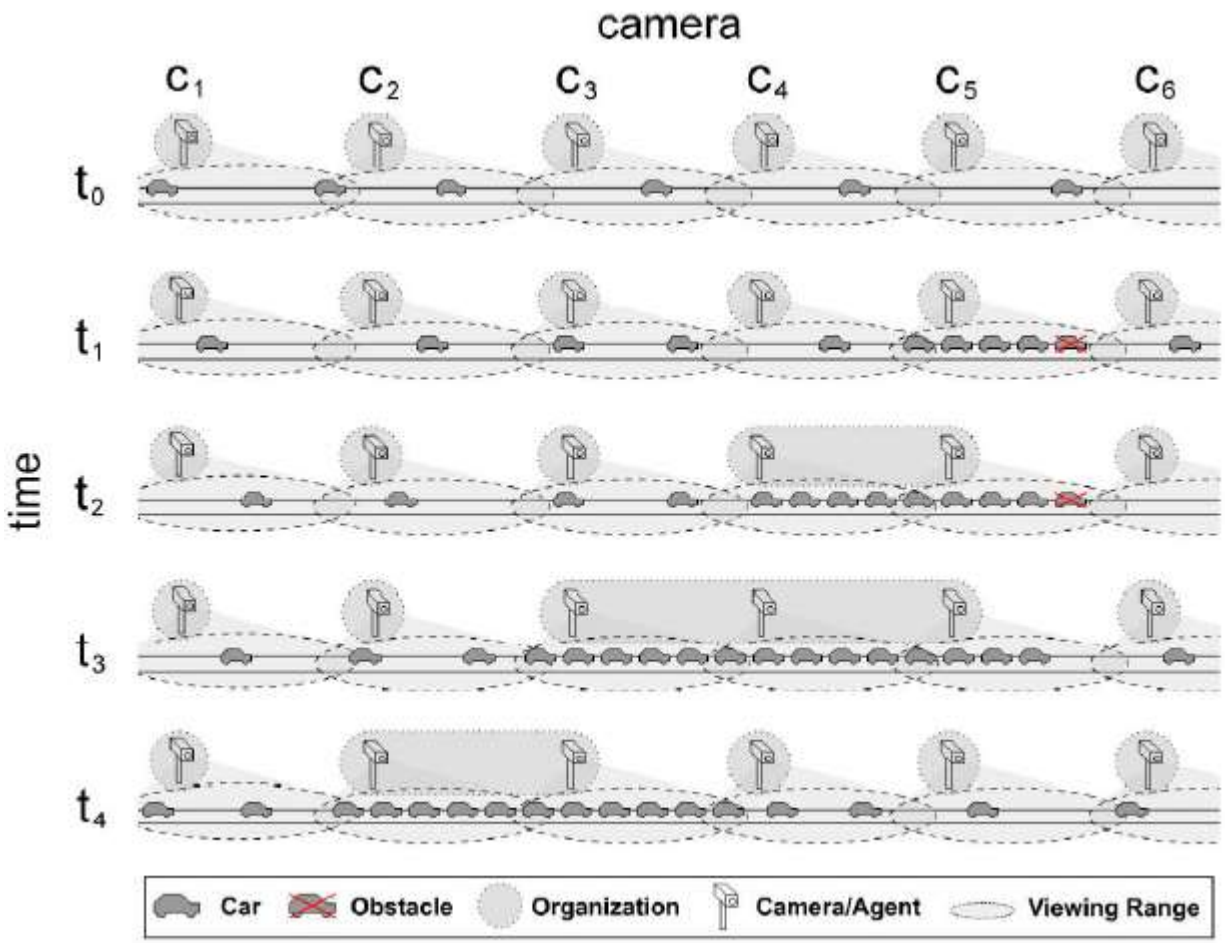


- Highway with traffic cameras
- 3 current traffic variables:
  - density (k = #car/length)
  - intensity (q = Σspeed/length)
  - average speed (u)
- Current congestion level (CCL) observed

Context-driven dynamic collaboration

- When traffic jam occurred (CCL passes threshold)
  - Camera collaborated with other cameras to capture the whole image of jam
  - Data are aggregated and reported
- When traffic jam propagated
  - Next *adjacent* camera will enter the collaboration
  - And push data to aggregator camera
- When traffic jam dissolved
  - Local camera will leave the collaboration
  - And keep observing CCL

System Scenario



Collaboration

- A sequence of cameras collaborate to determine jam
- Two roles in the group
  - Aggregator: aggregate data observed in the group (a single camera in the group)
  - Pusher: push data to the aggregator (all cameras except aggregator)

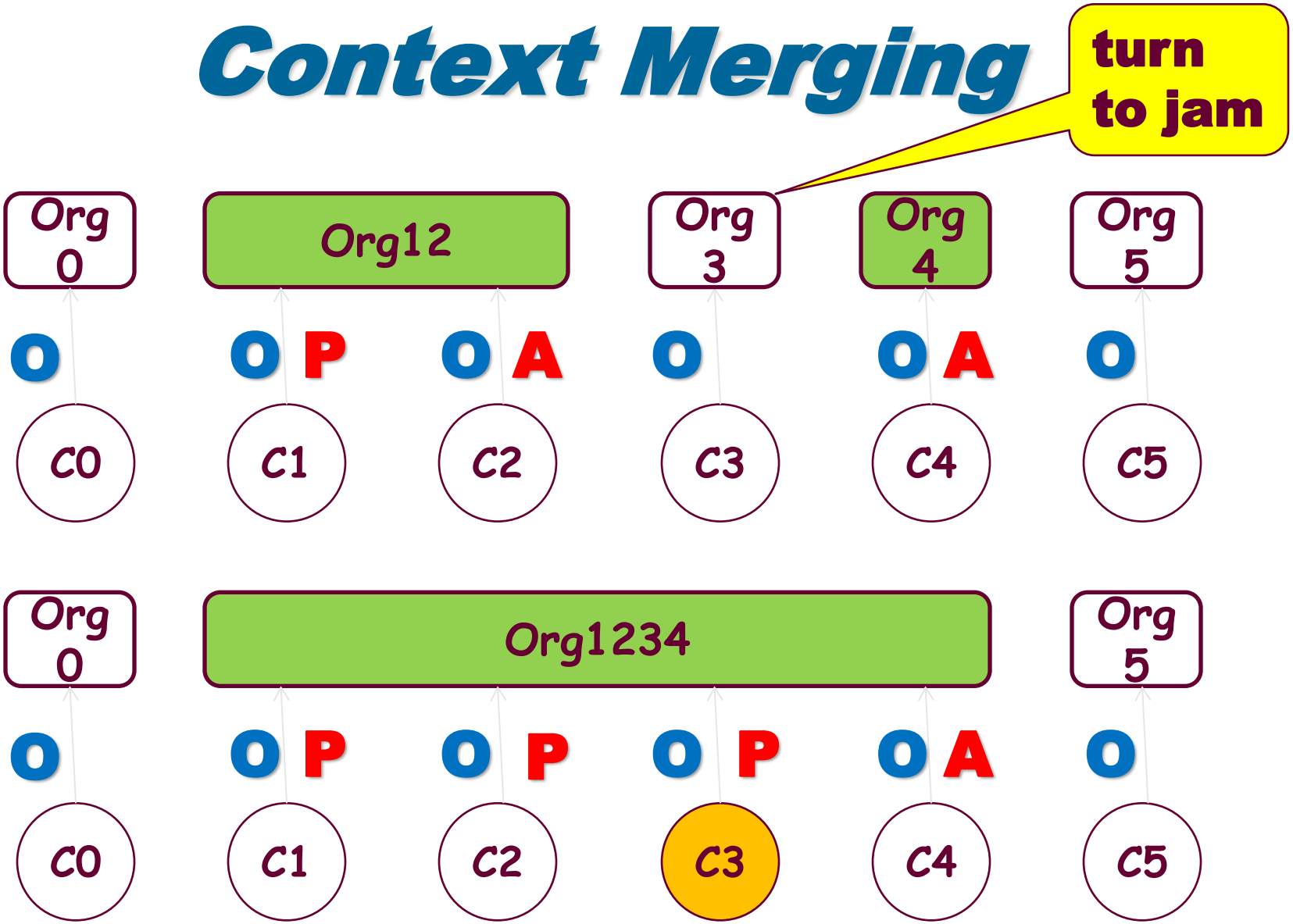
EpsilonJ Code Fragment

```
context Organization {
  role Client requires {void notify(Data);} {}
  role DataAggregator {
    Data amount;
    Data aggregate(Data d) {
      amount.append(d);}
    Data report() {
      Client.notify(amount);}}
  role DataPusher requires DataObserver {
    push(Data d) {
      DataAggregator.aggregate(getData());}}}
```

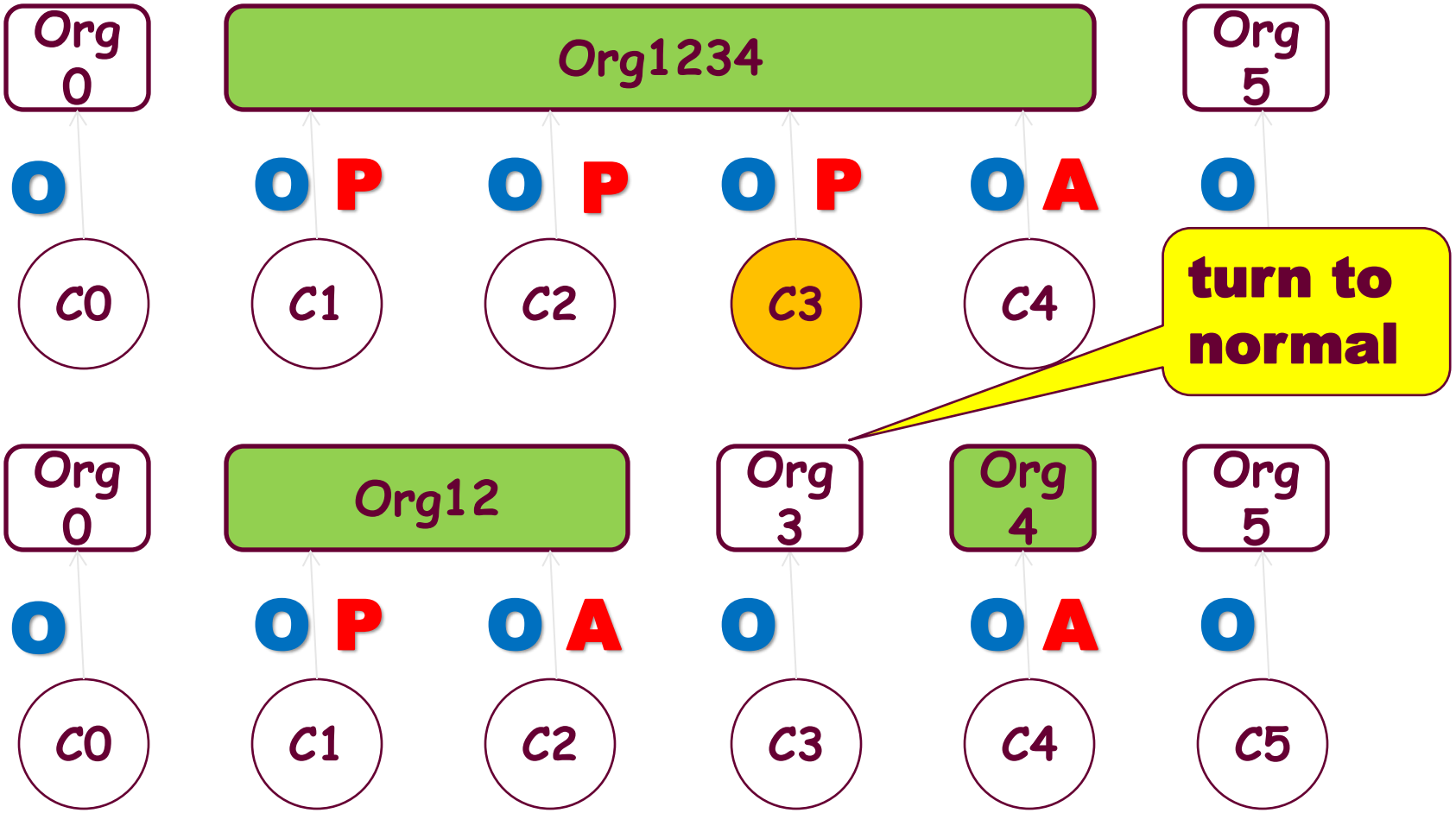
Context Merging/Splitting

- Context is
  - created when a new jam is detected;
  - killed when the jam is resolved;
  - merged when two jams join;
  - separated when jam is resolved in the middle.
- Role assignment changes after merging/splitting.

Context Merging



Context Splitting



EpsilonJ Code Fragment (2)

```
interface DataObserver {
  void observe(Data d);
  Data getData();}
class Camera implements DataObserver {
  Data current; Camera left, right;
  Organization org;
  void observe(Data d) {
    if(!current.inJam() && d.inJam()) {
      turnToJam();
    } else if(current.inJam() && !d.inJam()) {
      turnToNormal():}
    current = d;} ……
```

EpsilonJ Code Fragment (3)

```
void turnToJam() {
/* 1) new independent jam
   2) extend left jam
   3) extend right jam
   4) merge left and right jams */}
void turnToNormal() {
/* 1) single jam
   2) shrink left jam
   3) shrink right jam
   4) split to left and right jams */}
```

Merging/splitting may be general phenomena

- Companies merge and split
- Swarming of bee colonies
- Language support may be worth considering.

Is this autonomic?

- Changes are anticipated.
- Benefit of writing in EpsilonJ:
  - explicit representation of roles
  - type constraint stipulated by the current role:
    - aggregator, pusher

Self may be self-contradictory*

- If future changes in requirements or environment are
  - anticipated, it is hard to call it adaptive or self*
  - totally un-anticipated, it is impossible to program the measures against them

Conclusion

- Construction of self-sustainable systems is a hard but challenging target.
- A promising approach is to combine:
  - requirements engineering
  - context sensitive architecture