[illegible]

> A use case is a bundle of behavior [illegible] the basic functionality and underlying data

What the system is

vs.

What the system does

> Use cases are a variable part of a software system: can be added or removed, but also can change

> [illegible]

> Use cases are comprehensible to all stakeholders, including [illegible]

> [illegible]

> [illegible]

[illegible]

Make a Car Equipment Insurance Policy

Basic Flow

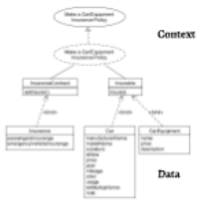
1. The insurer selects to make an insurance policy for car equipment.
2. The system prompts the insurer to prepare the insurance contract by filling in the required data.
3. The insurer fills in the information and submits it.
4. The system creates the insurance contract and asks for confirmation.
5. The insurer confirms the contract.
6. The use case ends.

Alternative Flow: Data Validation Error

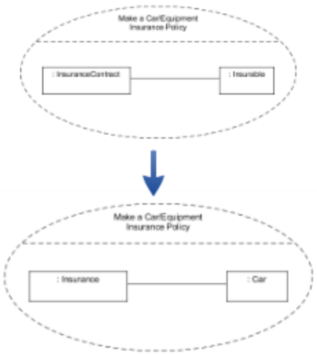
If the data entered in step 3 of the basic flow are not valid:

1. The system displays an error message indicating the nature of error.
2. The use case continues with step 3.

```
[illegible]
public interface InsuranceContext extends TransientComposite {
[illegible]

    abstract class Mixin implements InsuranceContext {

        /* Roles */
        [illegible]

        /* Context initialization */
        [illegible]
        }

        /* Context interaction */
        [illegible]
        }
    }
}
```



Interaction


Make a CarEquipment Insurance Policy
: InsuranceContract
: Insurable
Make a CarEquipment Insurance Policy
: Insurance
: Car

Summary

> The DCI approach decouples use cases as a (more) variable part of a software system from the underlying architecture (the system foundations)

> The Qi4j framework (now Apache Zest) enables to use DCI in Java

> A study of implementing DCI in Qi4j has been performed (a small car dealer system)

> Some conceptual and implementation specific observations with respect to the complexity of the realization have been reported here

vranic@stuba.sk
fiit.sk/~vranic

> Change requests are expressed in the application domain terms: the language of use cases

> With respect to use cases, any change request can be seen as a set of the following actions:

- Add a use case
- Remove a use case
- Alter a use case

> The evaluation of the approach has been performed qualitatively on the online shop application in terms of these actions

> The resulting changes to the code are well localized:

- Typically, only a few modules have to be changed
- In case of removal, modules are mostly removed as a whole

Assessing the DCI Approach to Preserving Use Cases in Code: Qi4J and Beyond

Jozef Zat'ko and Valentino Vranić

Institute of Informatics and Software Engineering

STU FIIT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF INFORMATICS
AND INFORMATION TECHNOLOGIES

zatko7071@gmail.com

vranic@stuba.sk
fiit.sk/~vranic

INES 2015

Bratislava, September 3, 2015

What is a use case and where is its place in the overall software system design?

Make a Car/Equipment Insurance Policy

Basic Flow

1. The insurer selects to make an insurance policy for a car or car equipment.
2. The system prompts the insurer to prepare the insurance contract by filling in the necessary data.
3. The insurer fills in the information and submits it.
4. The system creates the insurance contract and asks for confirmation.
5. The insurer confirms the contract
6. The use case ends.

Alternative Flow: Data Validation Error

If the data entered in step 4 of the basic flow are not valid:

1. The system displays an error message indicating the nature of error.
2. The use case continues with step 3.

Make a Car/Equipment Insurance Policy

Basic Flow

1. The insurer selects to make an insurance policy for a car or car equipment.
2. The system prompts the insurer to prepare the insurance contract by filling in the necessary data.
3. The insurer fills in the information and submits it.
4. The system creates the insurance contract and asks for confirmation.
5. The insurer confirms the contract
6. The use case ends.

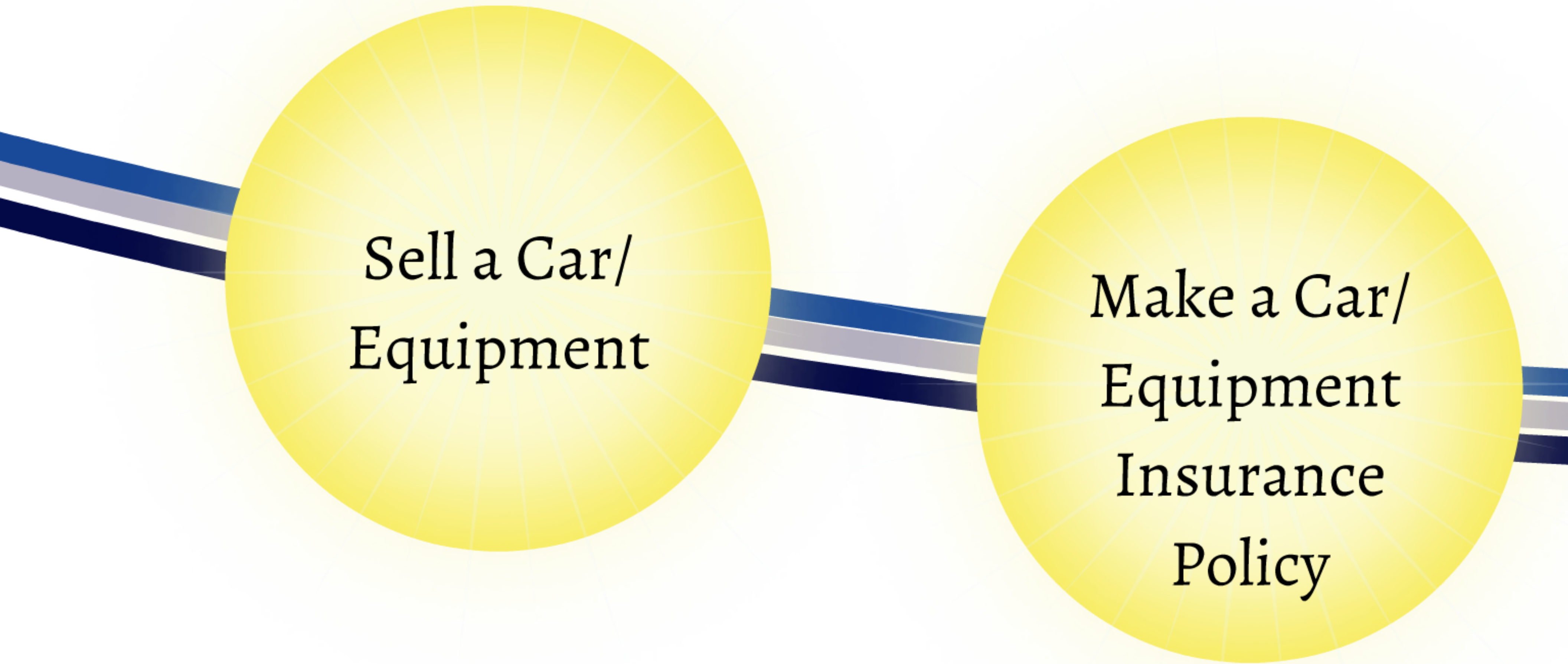
Alternative Flow: Data Validation Error

If the data entered in step 4 of the basic flow are not valid:

1. The system displays an error message indicating the nature of error.
2. The use case continues with step 3.

```
@Mixins(InsuranceContext.Mixin.class)
public interface InsuranceContext extends TransientComposite {
Logger LOG = Logger.getLogger(InsuranceContext.class);
    public void initContext(InsuranceContractRole insurance, InsurerRole insurer, InsurableRole insurable);
    public void executeContext();

    abstract class Mixin implements InsuranceContext {

        /* Roles */
        InsuranceContractRole insurance;
        InsurerRole insurer;
        InsurableRole insurable;

        /* Context initialization */
        public void initContext(InsuranceContractRole insurance, InsurerRole insurer, InsurableRole insurable) {
            LOG.info("InsureContext initialization");
            this.insurance = insurance;
            this.insurer = insurer;
            this.insurable = insurable;
        }

        /* Context execution */
        public void executeContext() {
            LOG.info("InsureContext execution");
            this.insurer.prepareInsuranceContract(insurance);
            this.insurance.setInsurer(insurer);
            this.insurable.insure(insurance);
            this.insurer.confirmInsuranceContract(insurance);
        }
    }
}
```

```
    this.insurer = insurer;
    this.insurable = insurable;
}

/* Context execution */
public void executeContext() {
    LOG.info("InsureContext execution");
    this.insurer.prepareInsuranceContract(insurance);
    this.insurance.setInsurer(insurer);
    this.insurable.insure(insurance);
    this.insurer.confirmInsuranceContract(insurance);
}
```



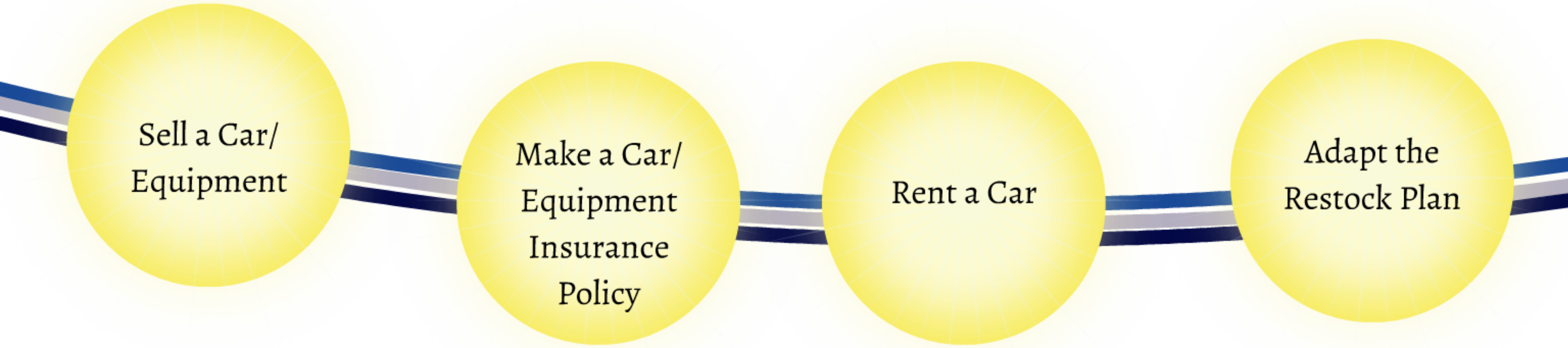
Sell a Car/ Equipment

Make a Car/ Equipment Insurance Policy

Sell a Car/ Equipment

Make a Car/ Equipment Insurance Policy

Rent a Car



Sell a Car/ Equipment

Make a Car/ Equipment Insurance Policy

Rent a Car

Adapt the Restock Plan

> A use case as a bead of behavior on the string of the basic functionality and underlying data

What the system is
vs.
What the system does

> Use cases are a variable part of a software system: can be added or removed, but also can change

> The underlying structure may change, too, but far less frequently

> Use cases are comprehensible to all stakeholders, including the users

> But once translated into code, a use case model quickly becomes outdated

> A need to retain/preserve use cases in the code itself

> What can be retained out of a use case in code?

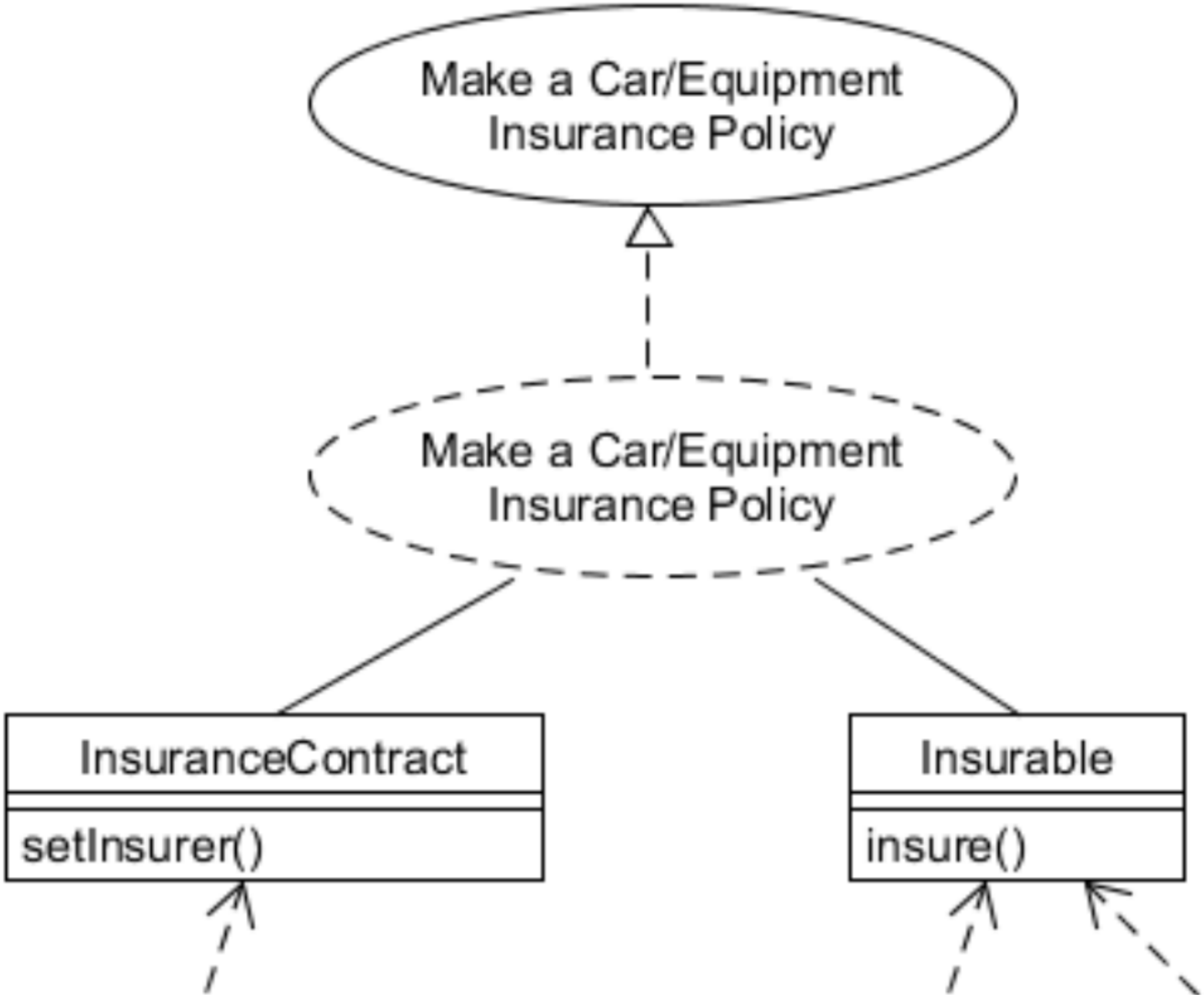
> Something is retained even unintentionally, but some approaches aim explicitly at preserving use cases in code

> DCI: Data, Context and Interaction (Reenskaug and Coplien)

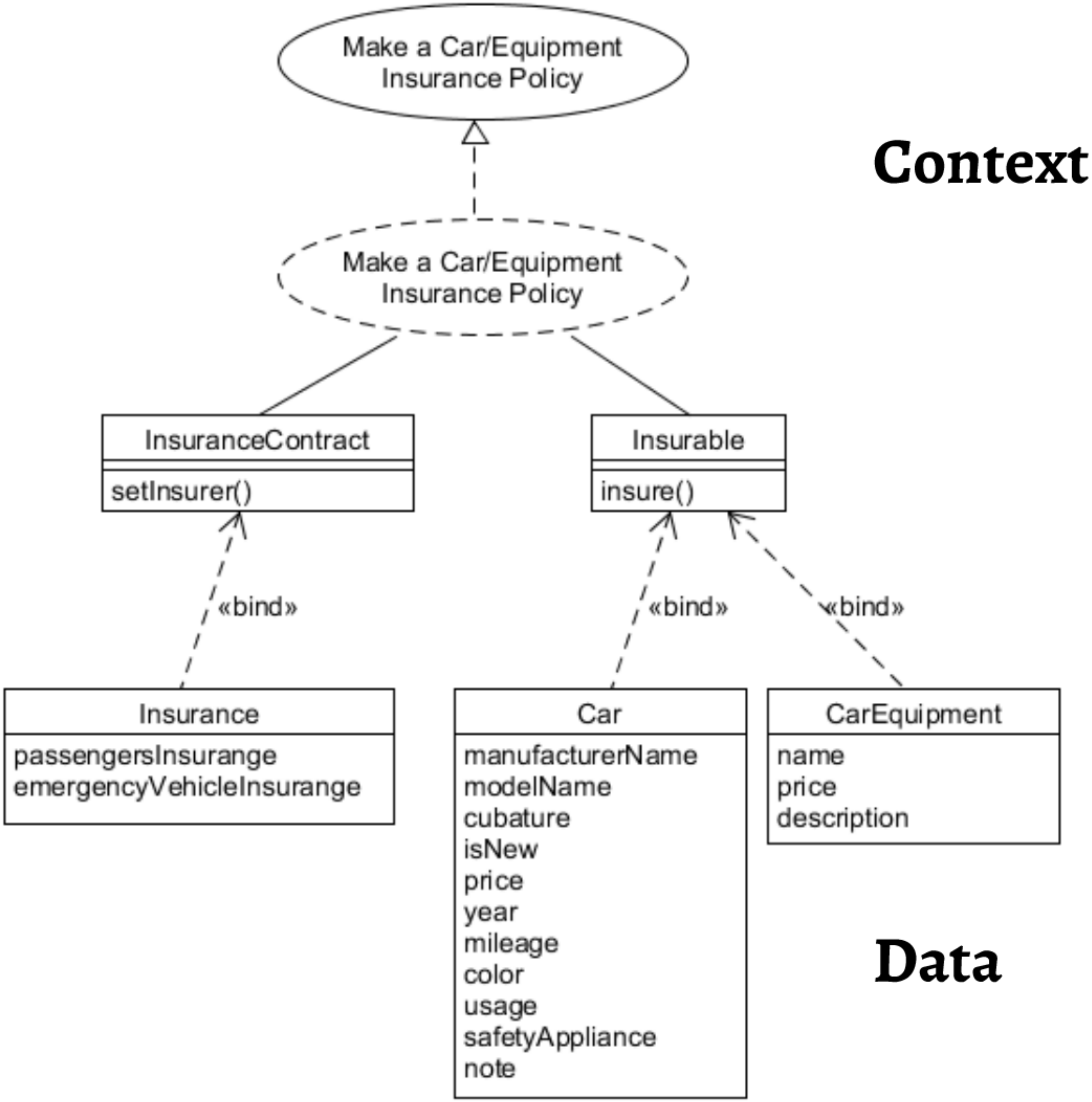
> Aspect-oriented software development with use cases (Jacobson and Ng)

> Preserving use case flows in source code (Bystrický and Vranić)

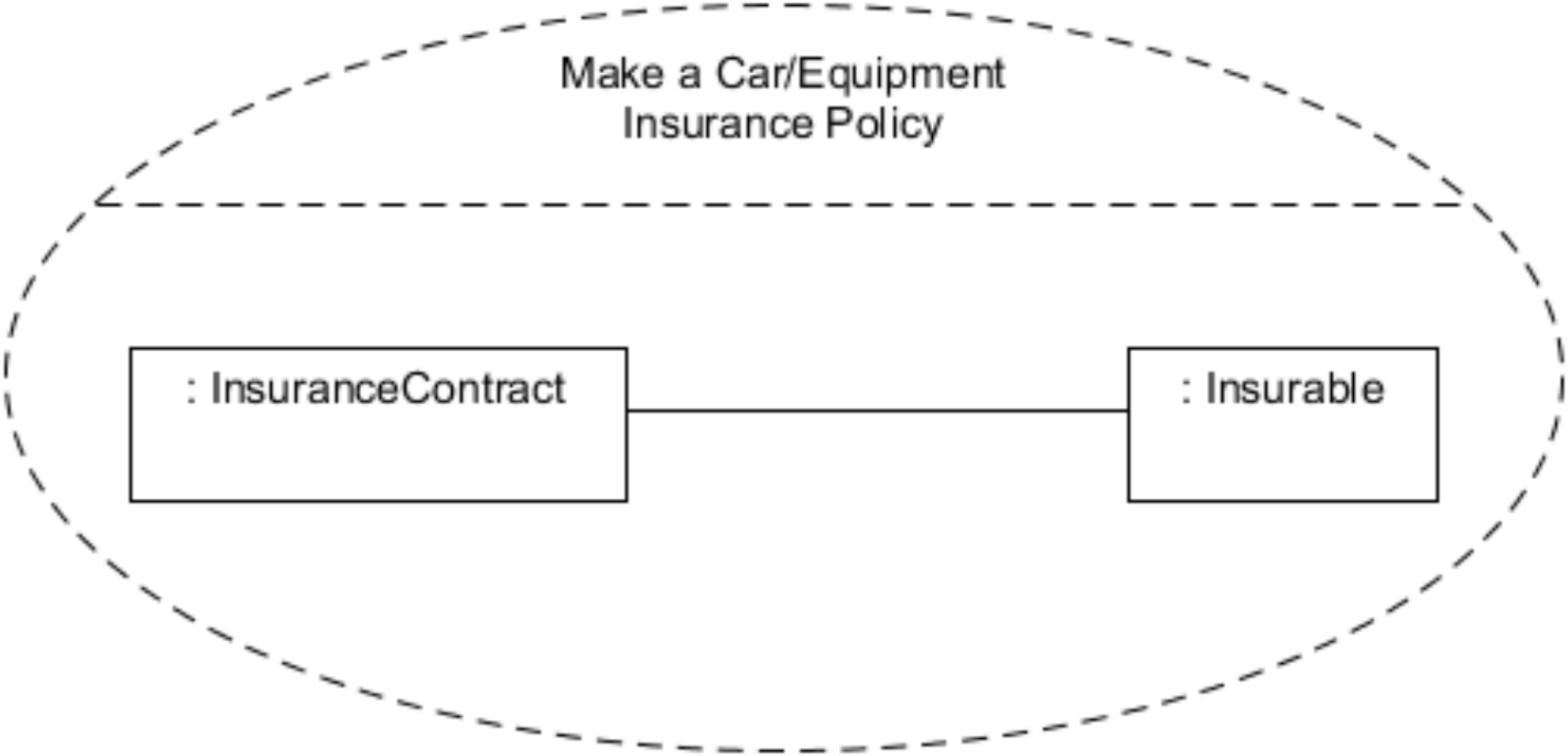
> An opportunistic approach to retaining use cases in OO source code (Greppel and Vranić)

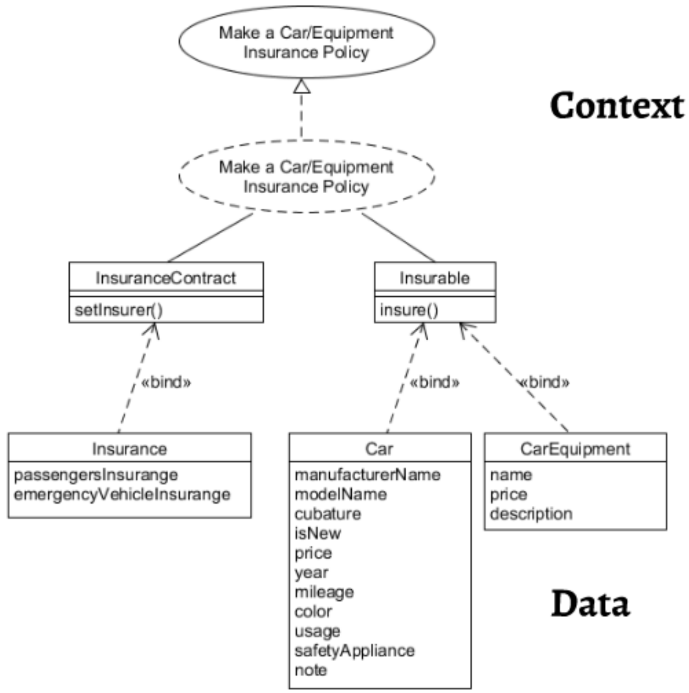

Make a Car/Equipment Insurance Policy
Make a Car/Equipment Insurance Policy
InsuranceContract
setInsurer()
Insurable
insure()

Context

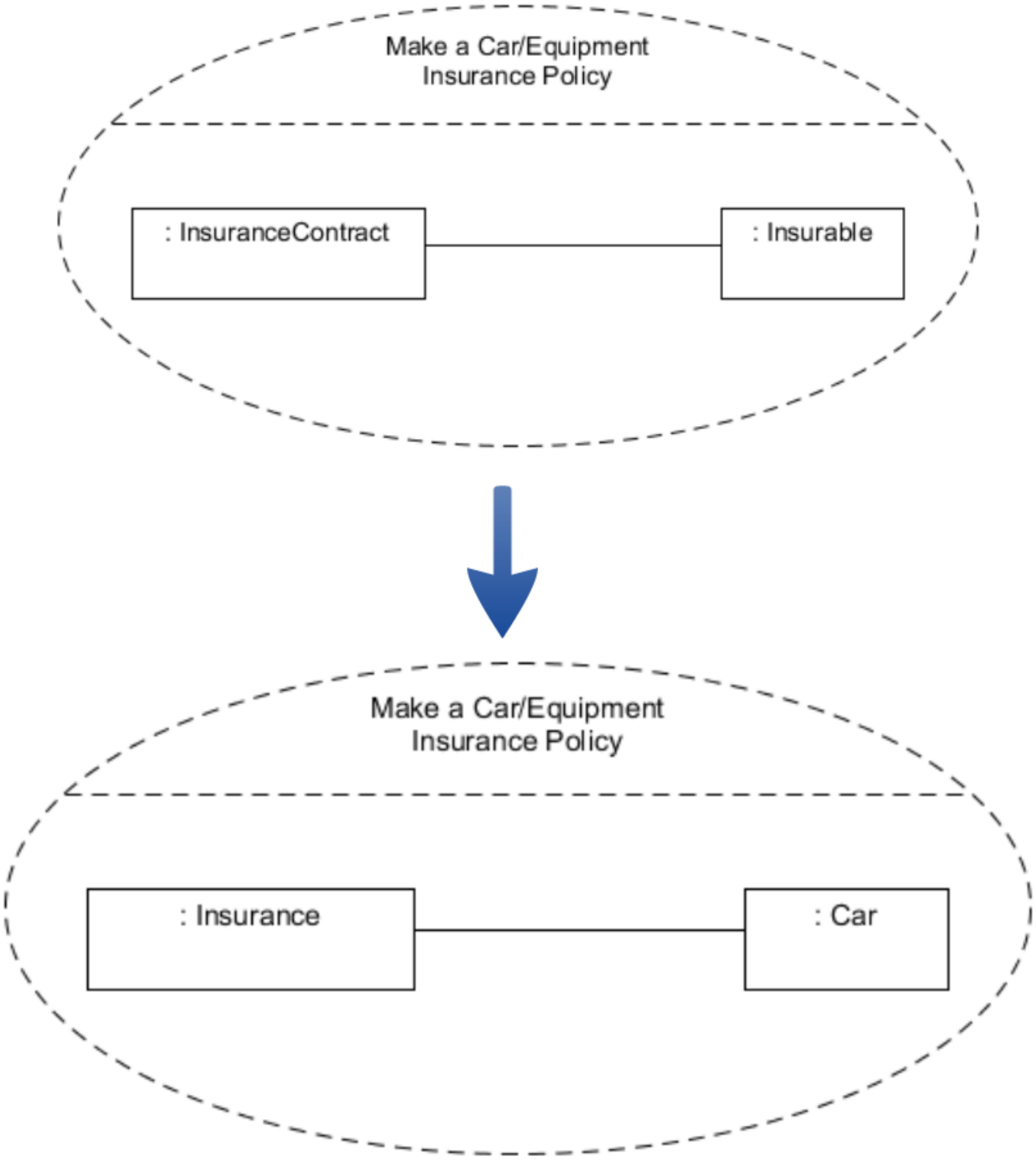

Make a Car/Equipment Insurance Policy
Context
Make a Car/Equipment Insurance Policy
InsuranceContract
setInsurer()
Insurable
insure()
«bind»
«bind»
«bind»
Insurance
passengersInsurange
emergencyVehicleInsurange
Car
manufacturerName
modelName
cubature
isNew
price
year
mileage
color
usage
safetyAppliance
note
CarEquipment
name
price
description
Data

Interaction





Interaction


Make a Car/Equipment Insurance Policy
: InsuranceContract
: Insurable
Make a Car/Equipment Insurance Policy
: Insurance
: Car

> The main point in DCI: use cases are expressed in terms of the roles the (data) objects play in them

> DCI needs a supporting mechanism for role binding

> The Qi4J framework (now Apache Zest) provides one for Java

> What we did:

- Implemented a small car dealer system in a DCI way using Qi4J

- Made some conceptual and implementation specific observations with respect to the complexity of the realization

Conceptual Observations

> Roles can reduce inheritance use and decrease maintainability effort

> Generic roles can be played by objects of inappropriate classes

In Qi4J, a direct access to the domain model from the generic context roles is lost

```
public interface InsuranceContractRole extends TransientComposite {
    ...
    /* The interface represents the data of the objects playing current role */
    public interface InsuranceData {
        /* The attributes from the data object */
        @Optional Property<Date> createDate();
        @Optional Property<Date> signDate();
        @Optional Property<InsurerRole> seller();
        @Optional Property<Boolean> isApproved();
        ...
    }
    abstract class Mixin implements InsuranceContractRole {
        @This
        InsuranceData data;
        public void setCreateDate(Date d) {  data.createDate().set(d);  }
        public void setSignDate(Date d) {  data.createDate().set(d);  }
        public void setInsurer(InsurerRole i) {  data.seller().set(i);  }
        public void setApproveFlag(Boolean b) {  data.isApproved().set(b);  }
        ...
```

In Qi4J, entities define their casting rules

```
public interface SellerEntity extends EntityComposite,
    //Data
    SellerData,
    //Roles
    ApproverRole,
    ContractorRole,
    InsurerRole
    { }
```

In Qi4J, interfaces have to be used instead of classes as templates for objects

```
public interface CarData {
    ...
    public String getName();
    abstract class Mixin implements CarData {
        /* Returns appended car name (manufacturer + model)  */
        public String getName() {
            StringBuffer sb = new StringBuffer("");
            sb.append(this.manufacturerName().get());
            sb.append(" ");
            sb.append(this.modelName().get());
            return sb.toString().trim();
        }
    }
}
```

In Qi4J, there is no access management of the data class attributes and methods

```
public interface CarData {
     // All public
    @Optional Property<ContractData> contract();
    @Optional Property<String> manufacturerName();
    @Optional Property<String> modelName();
    @Optional Property<String> color();
    @Optional Property<Boolean> isNew();
    @Optional Property<Double> price();
    @Optional Property<Integer> year();
    ...
}
```

In Qi4J, there is no direct support of polymorphism

```
@Mixins(InsurableRole.Mixin.class)
public interface InsurableRole extends TransientComposite {
    public void insure(InsuranceContractRole insurance);
    /* The interface represents the data of the objects playing current role */
    interface Data {
        @Optional Property<Double> price();
        @Optional Property<Boolean> isNew();
    }
    abstract class Mixin implements InsurableRole {
        @This
        Data data;
        public void insure(InsuranceContractRole insurance) {
            if (data.isNew()!=null) {
                insurance.setAnnualPayment(data.price().get());
            }
            else {
                insurance.setAnnualPayment(new Double(0.0));
            }
        }
    }
}
```

Summary

> The DCI approach decouples use cases as a (more) variable part of a software system from the underlying architecture (the system foundations)

> The Qi4J framework (now Apache Zest) enables to use DCI in Java

> A study of implementing DCI in Qi4J has been performed (a small car dealer system)

> Some conceptual and implementation specific observations with respect to the complexity of the realization have been reported here

vranic@stuba.sk
fiit.sk/~vranic