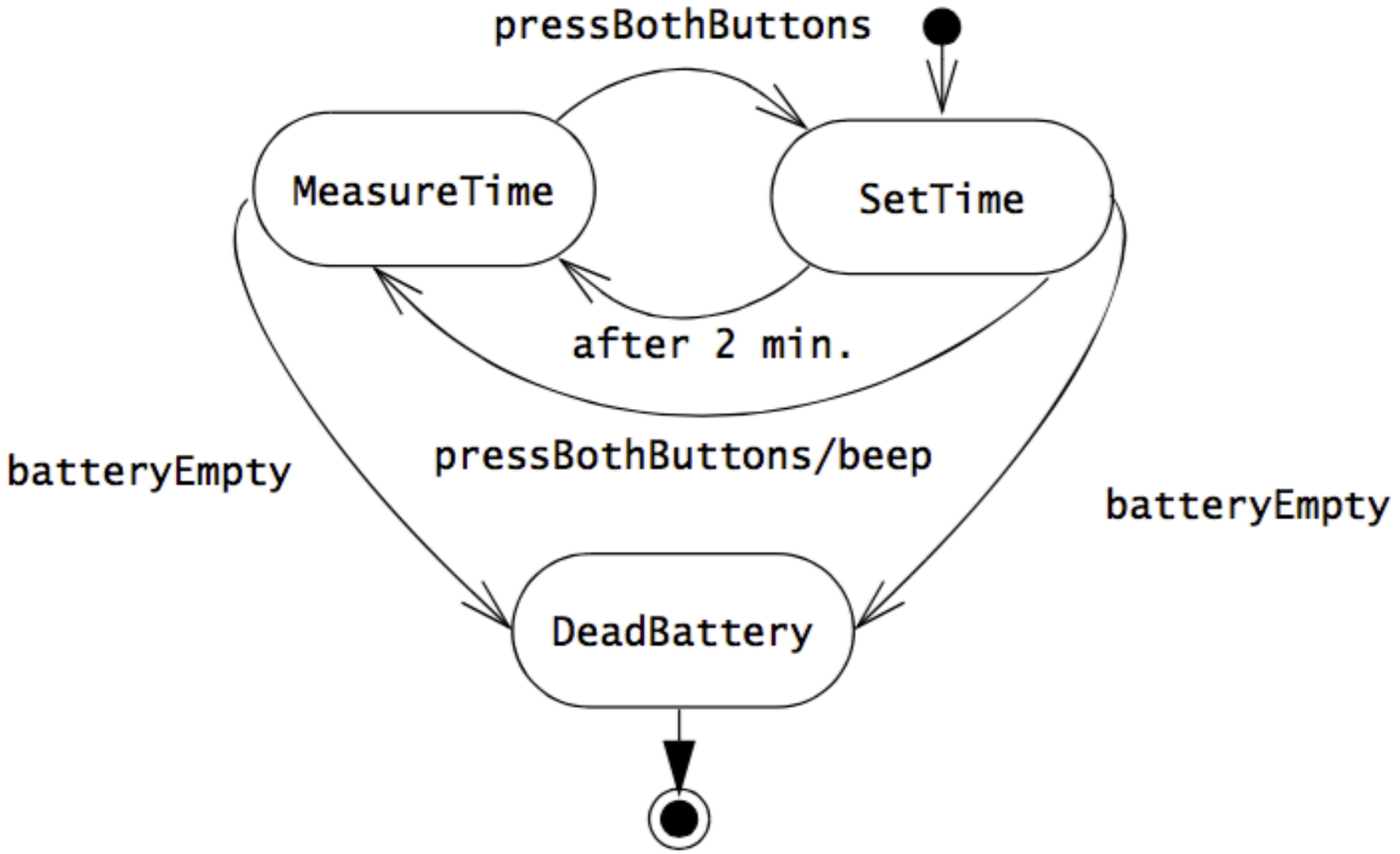
Analysis, Design, and Software Architecture (BDSA)

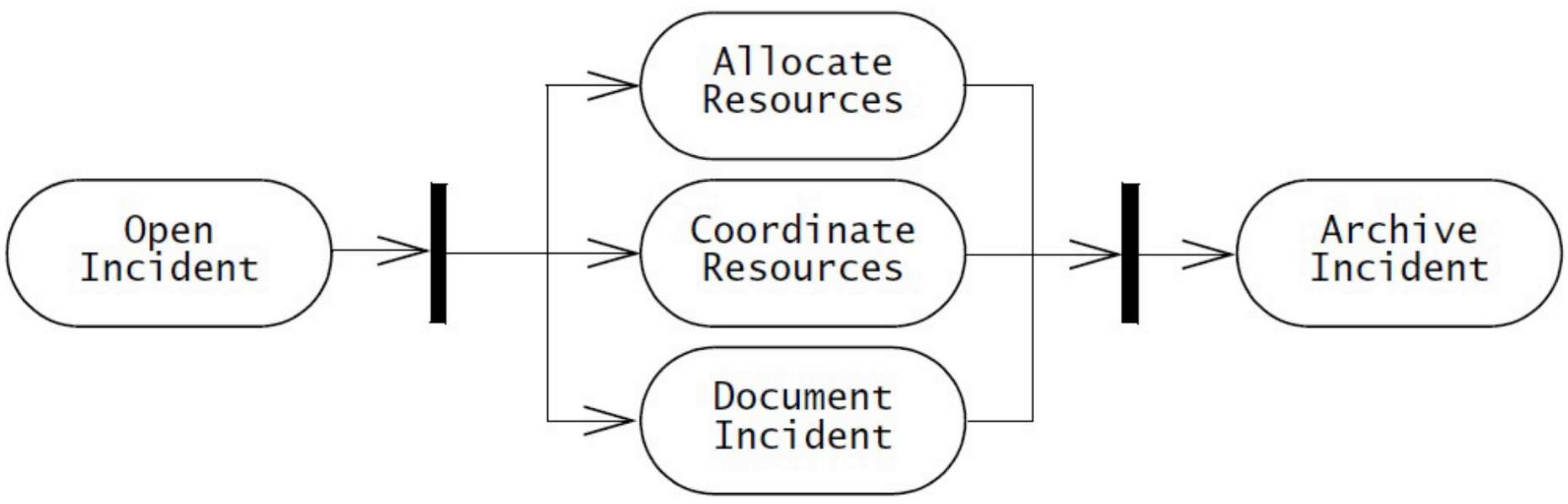
Paolo Tell

SOLID principles

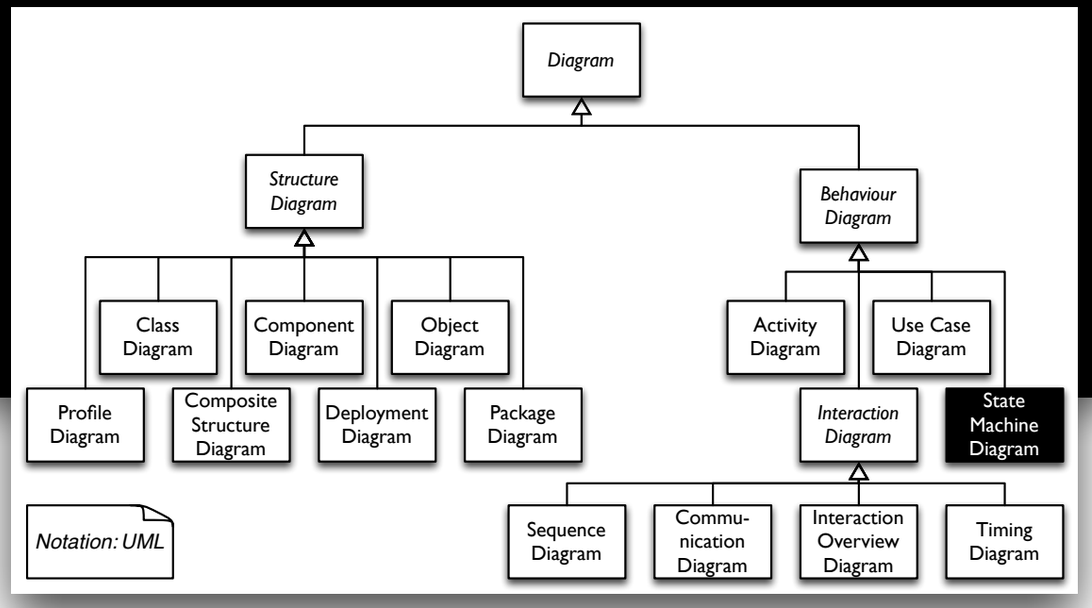
IT University
of Copenhagen

Recap

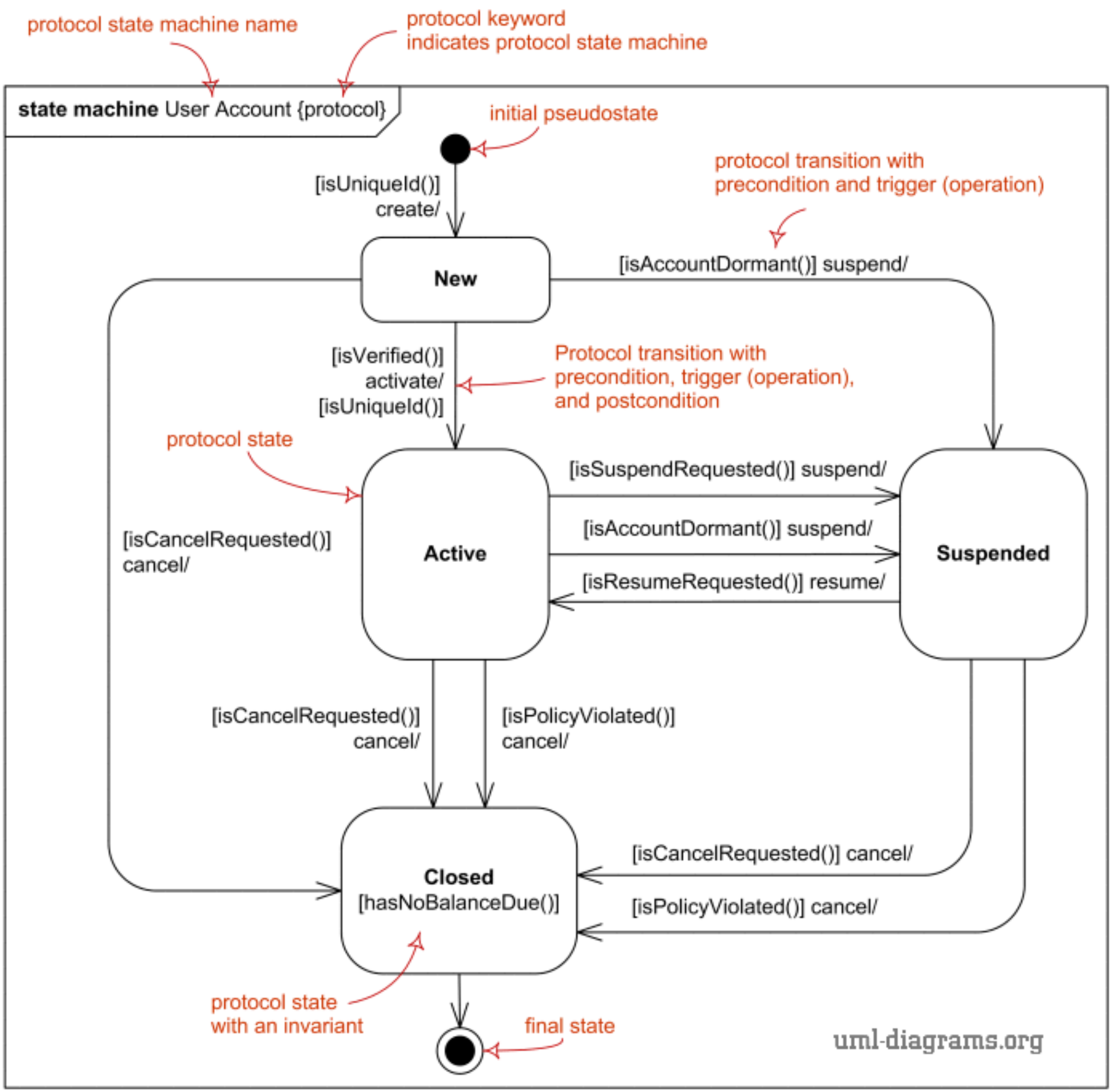

pressBothButtons
MeasureTime
SetTime
after 2 min.
batteryEmpty
pressBothButtons/beep
batteryEmpty
DeadBattery


Open Incident
Allocate Resources
Coordinate Resources
Document Incident
Archive Incident

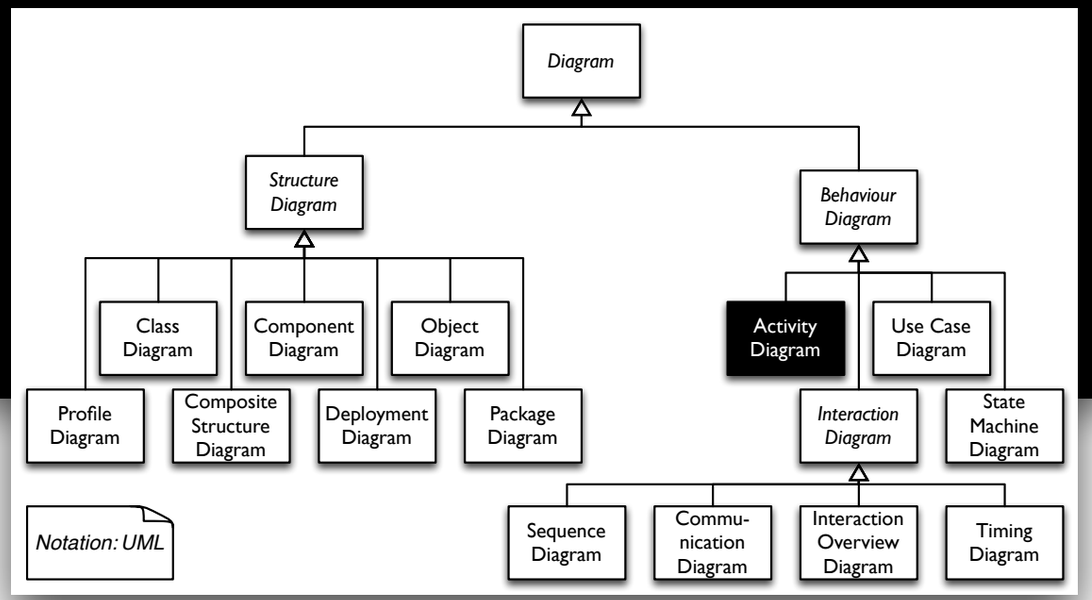
State machine diagram



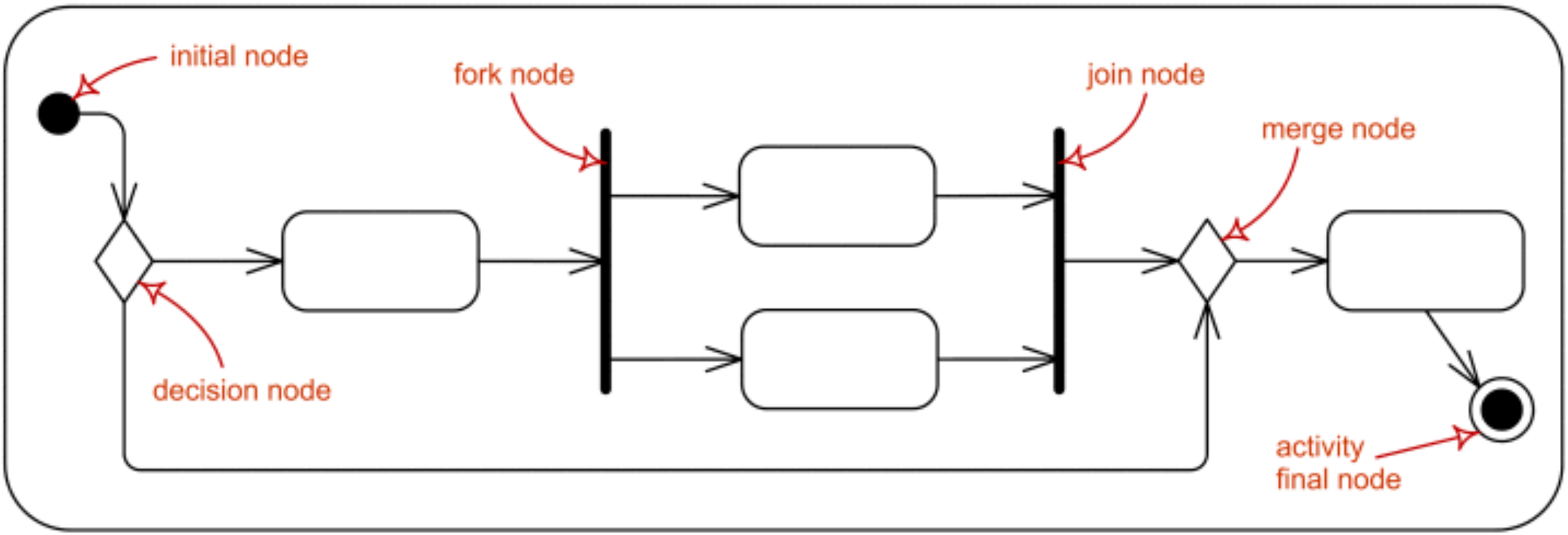
- Behaviour diagrams.
- They specify the dynamic behaviour of a single object.
- They model the sequence of states an object goes through at runtime in reaction to external events.
- They include an initial and final state.
- They show states and transactions.
- Substates are permitted.
- Transaction are labeled with events, guards, etc.

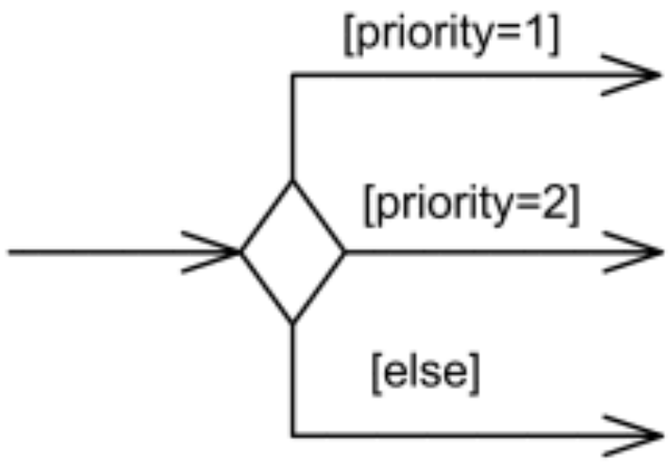

protocol state machine name
protocol keyword
indicates protocol state machine
state machine User Account {protocol}
initial pseudostate
[isUniqueId()]
create/
protocol transition with
precondition and trigger (operation)
New
[isAccountDormant()] suspend/
[isVerified()]
activate/
[isUniqueId()]
Protocol transition with
precondition, trigger (operation),
and postcondition
protocol state
[isSuspendRequested()] suspend/
[isCancelRequested()]
cancel/
Active
[isAccountDormant()] suspend/
Suspended
[isResumeRequested()] resume/
[isCancelRequested()]
cancel/
[isPolicyViolated()]
cancel/
[isCancelRequested()] cancel/
Closed
[hasNoBalanceDue()]
[isPolicyViolated()] cancel/
protocol state
with an invariant
final state
uml-diagrams.org

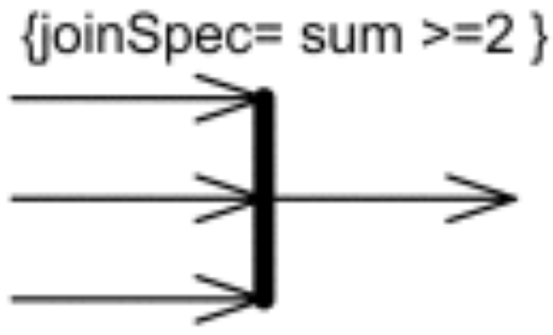
Activity diagram



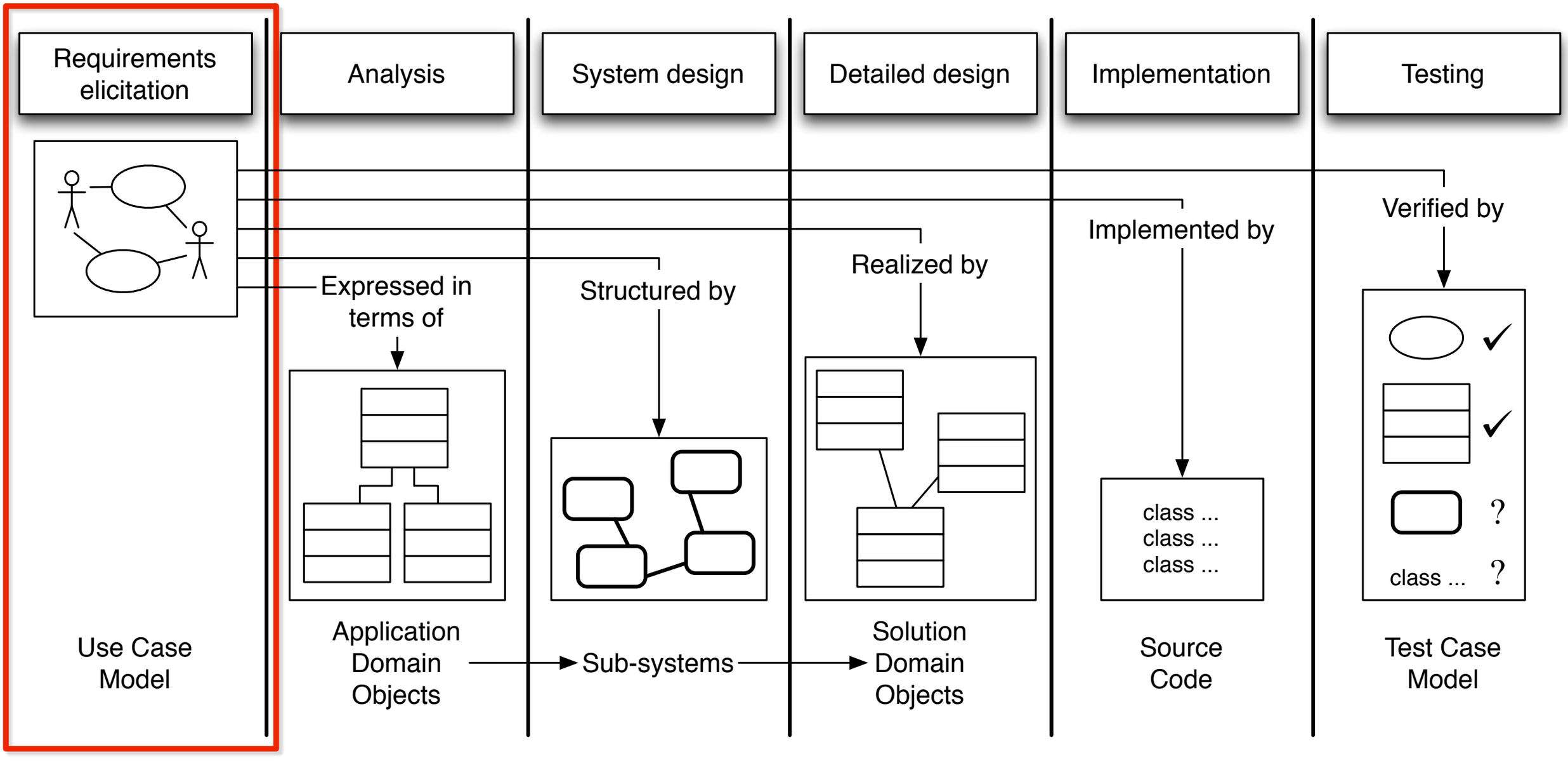
- Behaviour diagrams.
- They model the dynamic behaviour of a sub-system.
- They focus on lower level behaviour.
- They are realized on terms of one or several sequences of activities.
- Also known as flowchart.
- They include an initial and final state.
- They show decisions/merges and forks/joins.
- They can be partitioned in swimlanes to highlight concerns.


initial node
fork node
join node
merge node
decision node
activity
final node

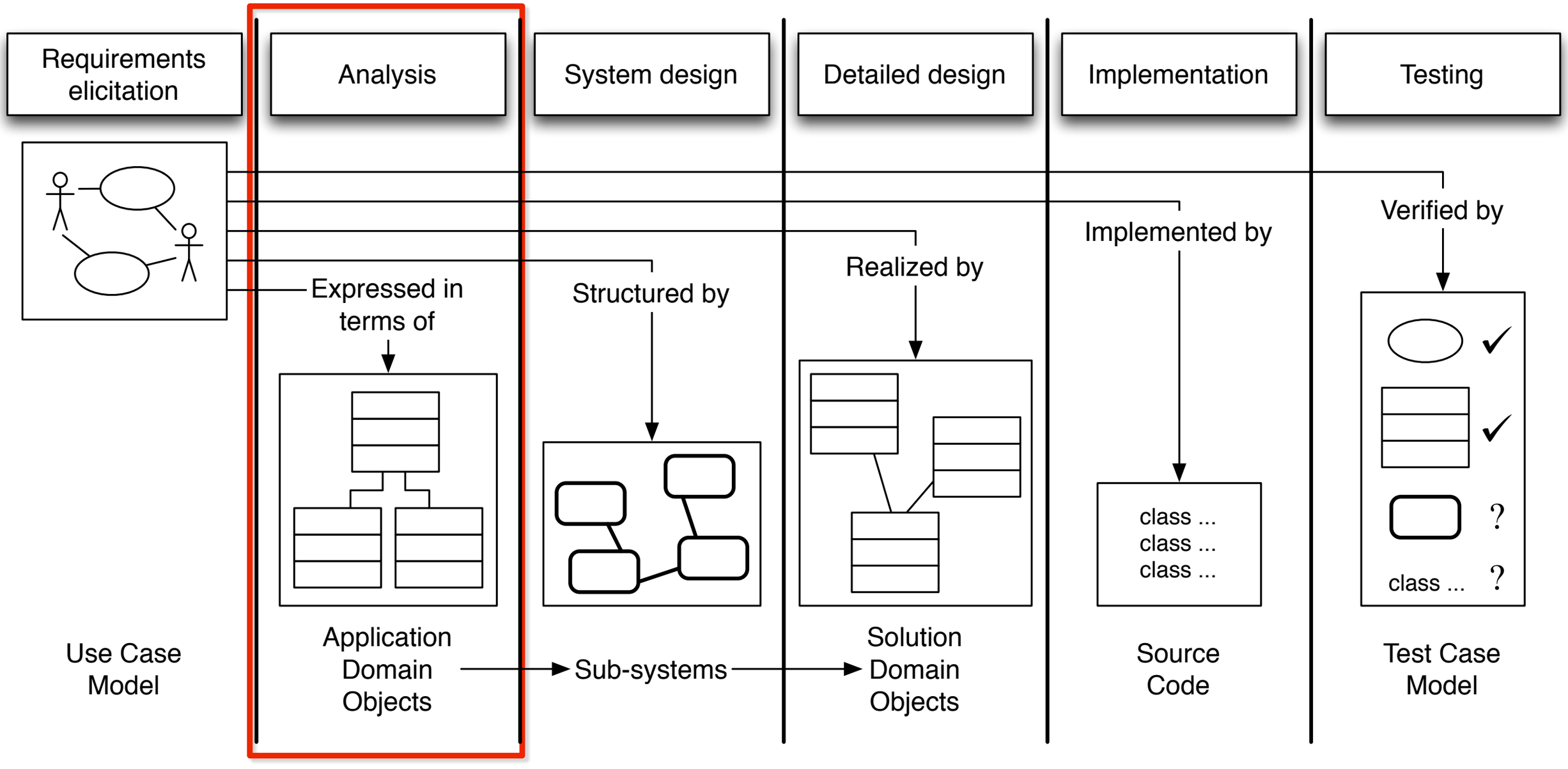

[priority=1]
[priority=2]
[else]


{joinSpec= sum >=2 }

Software lifecycle activities



Software lifecycle activities

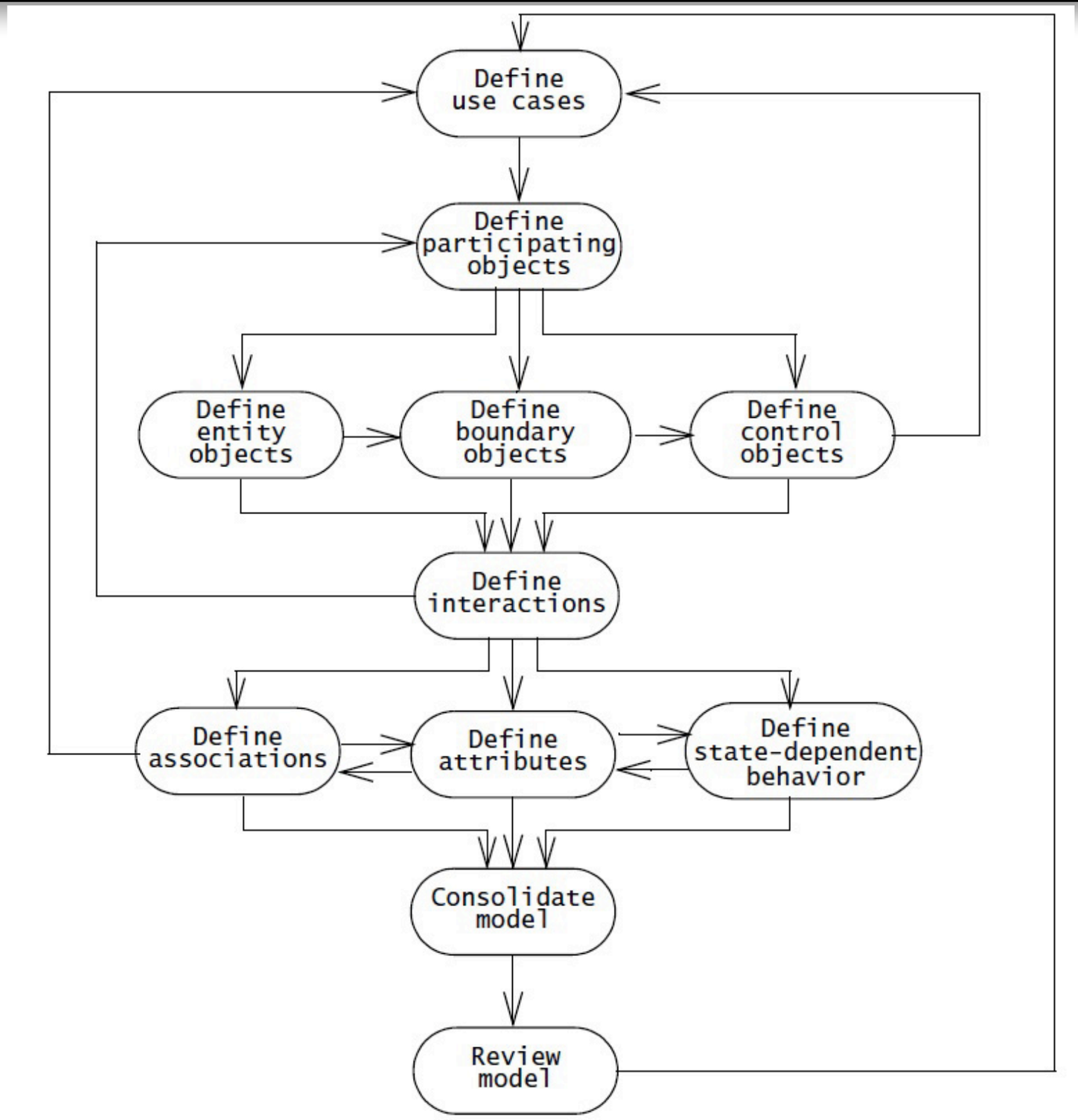


Object-oriented analysis

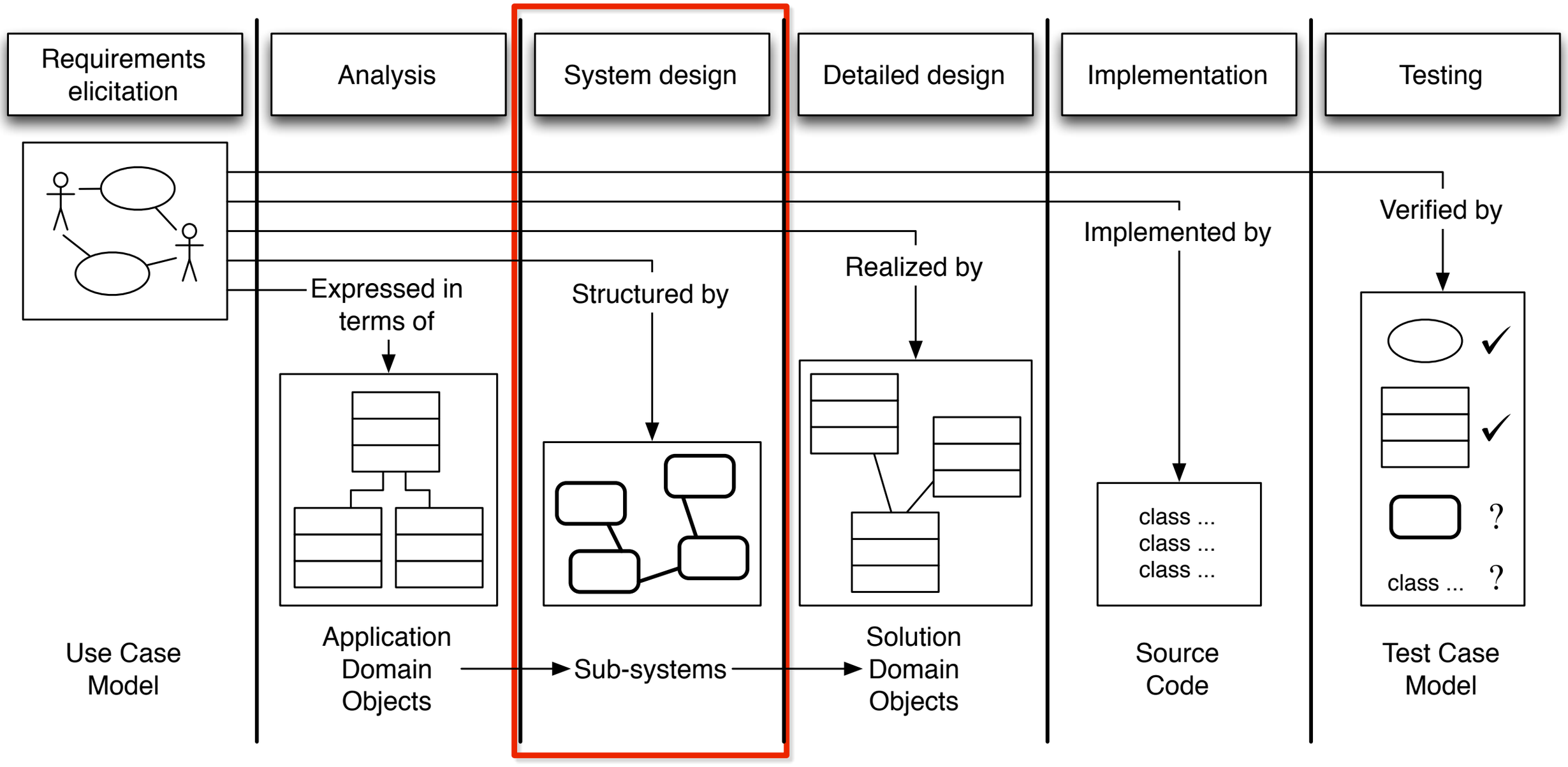
- OOA focuses on creating a model of the system:
  - that is complete, correct, consistent, and verifiable;
  - by structuring formalizing requirements;
  - which leads to revision of the requirements.

- Analysis model consists of:
  - functional model – use case model;
  - analysis object model – class & object diagrams;
  - dynamic model – sequence & state machine diagrams.

Analysis activities



Software lifecycle activities



Outline

- Literature
  - [OOSE] ch. 6
  - [SE9] ch. 6

- Topics covered:
  - SOLID principles

S.O.L.I.D.

Symptoms of bad software

- Confusing. A good software should explain what it is doing.

1.

2.

3.

Symptoms of bad software

- Confusing. A good software should explain what it is doing.

1. Rigidity. To make a modification, you need “touch” a lot of other stuff.
2.
3.

Symptoms of bad software

- Confusing. A good software should explain what it is doing.

1. Rigidity. To make a modification, you need "touch" a lot of other stuff.

2. Fragility. You change something somewhere, and it breaks the software in unexpected areas that are entirely unrelated to the change you made.

Symptoms of bad software

- Confusing. A good software should explain what it is doing.

1. Rigidity. To make a modification, you need "touch" a lot of other stuff.

2. Fragility. You change something somewhere, and it breaks the software in unexpected areas that are entirely unrelated to the change you made.

3. Immobility (no-reusability). The software does more than what you need. The desirable parts of the code are so tightly coupled to the undesirable ones that you cannot use the desirable parts somewhere else.

Symptoms of bad software

- Confusing. A good software should explain what it is doing.

1. Rigidity. To make ...er stuff.
2. Fragility. You char... software in unexpected areas ...u made.
3. Immobility (no-re... ...at you need. The desirable parts of the code are so tightly coupled to the undesirable ones that you cannot use the desirable parts somewhere else.

What do these problems have in common?

Symptoms of bad software

- Confusing. A good software should explain what it is doing.

1. Rigidity. To make ...er stuff.
2. Fragility. You char... software in unexpected areas ...u made.
3. Immobility (no-re... ...at you need. The desirable parts of the code are so tightly coupled to the undesirable ones that you cannot use the desirable parts somewhere else.

What do these problems have in common?

Spaghetti code - Coupling - Dependencies

Symptoms of bad software

- Confusing. A good software should explain what it is doing.

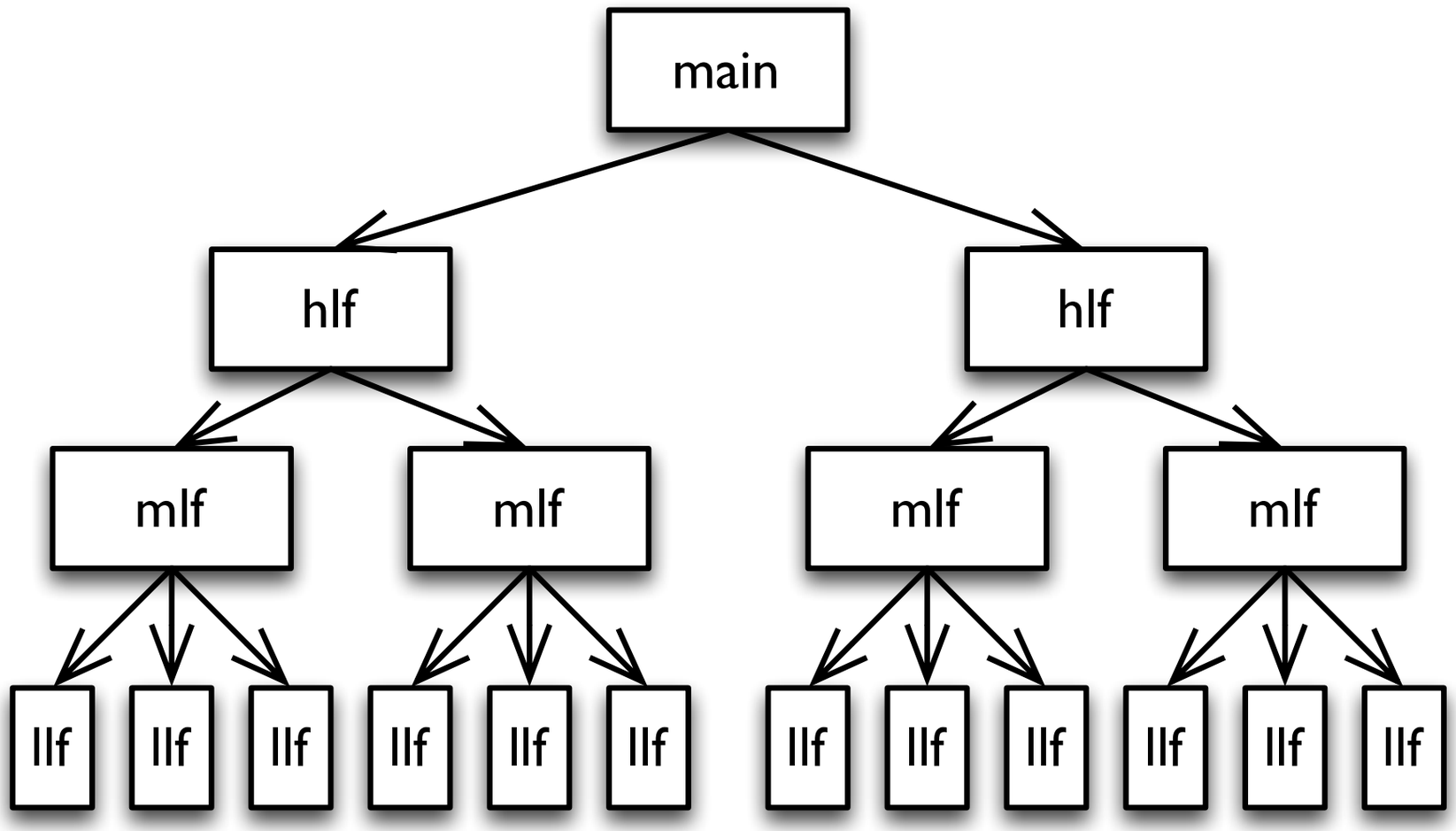
1. Rigidity. To make
2. Fragility. You char
unexpected areas
3. Immobility (no-re
desirable parts of the code are so tightly coupled to the undesirable ones that you cannot use the desirable parts somewhere else.

What do these problems have in
er stuff.
software in
u made.
t you need. The

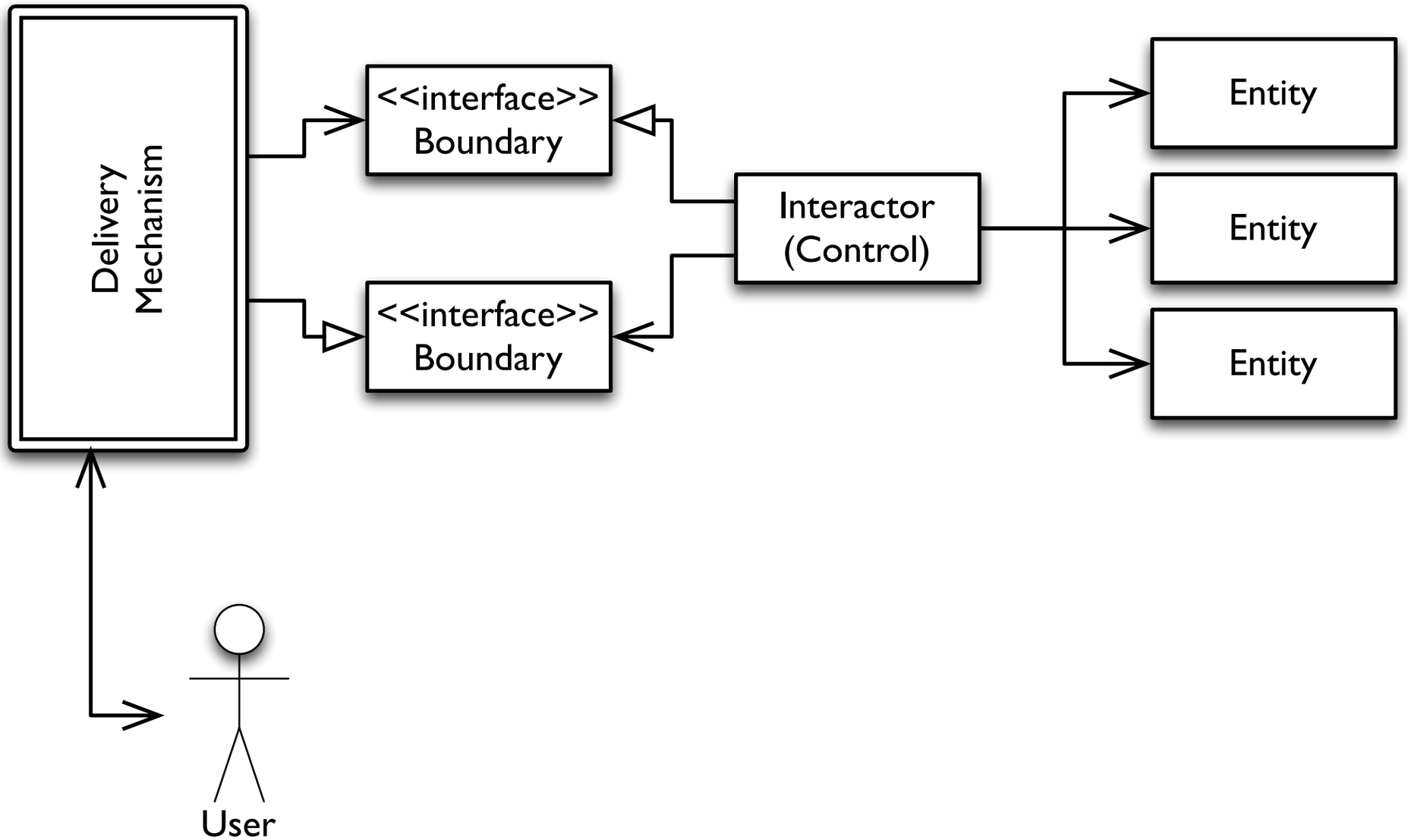
How do you do that?

How do you manage your dependencies?

In (procedural) structured programming



Interactor (Control) - Entity - Boundary

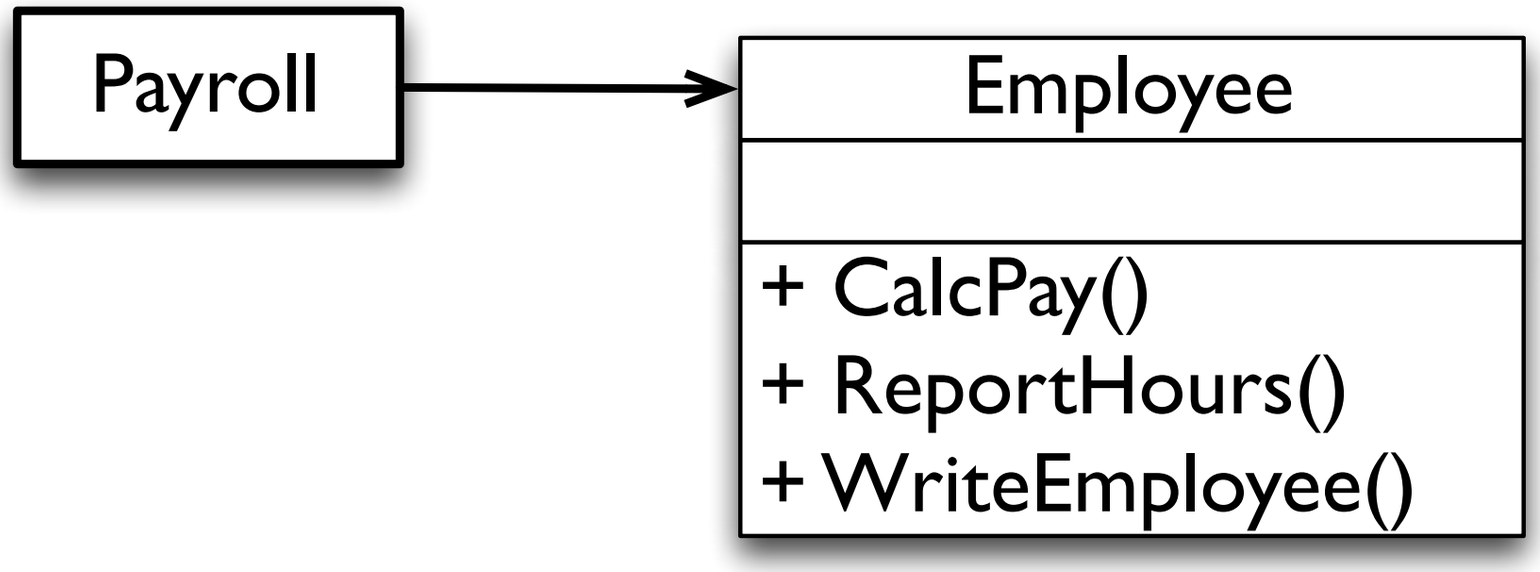


Robert Cecil Martin (uncle Bob)

S.O.L.I.D.

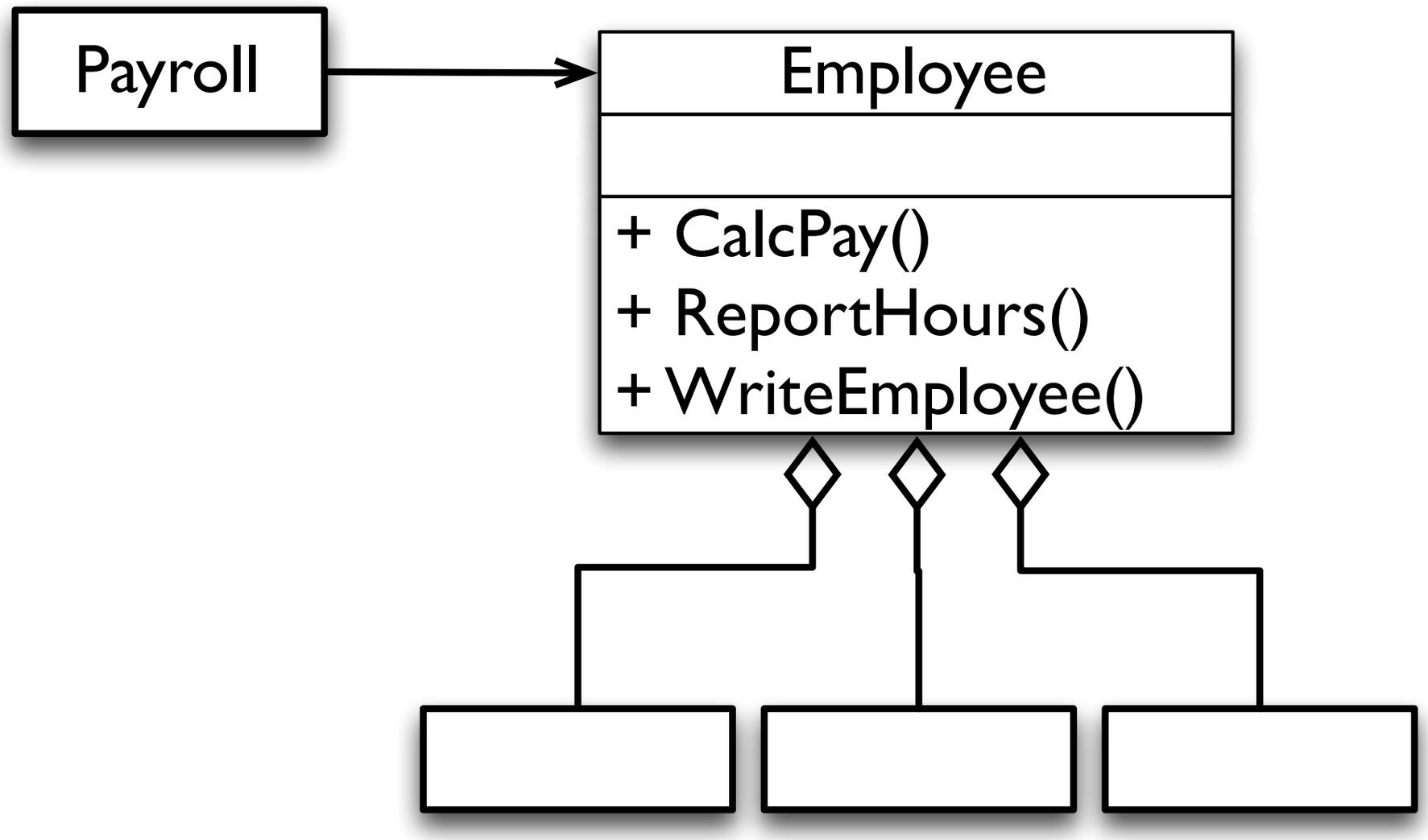
- Single responsibility principle: “a class should only have one, and only one, reason to change”.
- Open/closed principle: “software entities should be open for extensions but closed for modifications”. (Bertrand Meyer 1988)
- Liskov substitution principle: “derived classes should be usable through the base class interface, without the need for the user to know the difference”. (Barbara Liskov 1987)
- Interface segregation principle: “many client-specific interfaces are better than one general-purpose interface”.
- Dependency inversion principle: “depend upon abstractions, do not depend upon concretions”.

Single responsibility principle



- A class should only have one, and only one, reason to change.
- A class should have only a single responsibility.

Single responsibility principle



Open/closed principle

Shape.h

```
enum ShapeType {circle, square};
struct Shape
     {enum ShapeType itsType;};
```

Circle.h

```
struct Circle
{
  enum ShapeType itsType;
  double itsRadius;
  Point itsCenter;
};
void DrawCircle(struct Circle*)
```

Square.h

```
struct Square
{
  enum ShapeType itsType;
  double itsSide;
  Point itsTopLeft;
};
void DrawSquare(struct Square*)
```

DrawAllShapes.c

```
#include <Shape.h>
#include <Circle.h>
#include <Square.h>

typedef struct Shape* ShapePtr;

void
DrawAllShapes(ShapePtr list[], int n)
{
  int i;
  for( i=0; i< n, i++ )
  {
    ShapePtr s = list[i];
    switch ( s->itsType )
    {
    case square:
      DrawSquare((struct Square*)s);
      break;
    case circle:
      DrawCircle((struct Circle*)s);
      break;
    }
  }
}
```

- Software entities should be open for extensions but closed for modifications.

Open/closed principle

Shape.h

```
Class Shape
{
public:
   virtual void Draw() const=0;
};
```

Square.h

```
Class Square: public Shape
{
public:
   virtual void Draw() const;
};
```

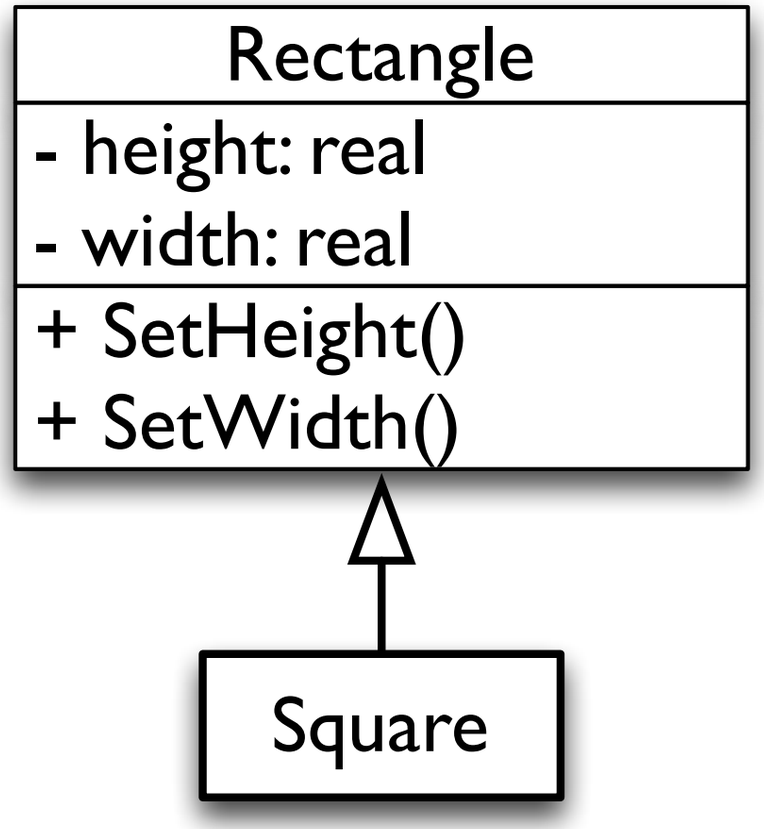
Circle.h

```
Class Circle: public Shape
{
public:
   virtual void Draw() const;
};
```

DrawAllShapes.cpp

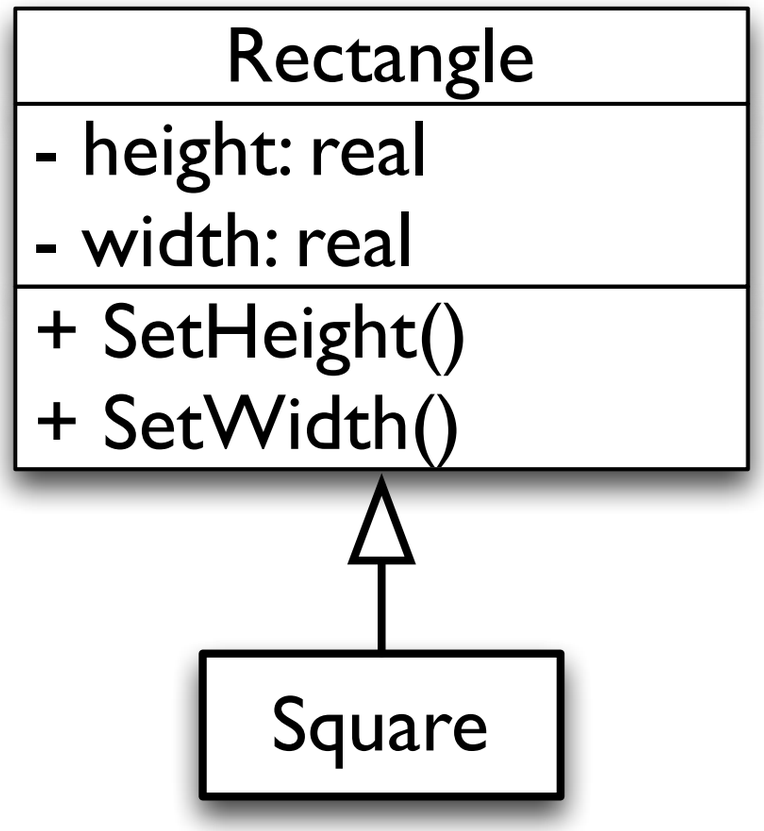
```
#include <Shape.h>

void
DrawAllShapes(Shape* list[],int n)
{
  for(int i=0; i< n; i++)
    list[i]->draw();
}
```

Liskov substitution principle



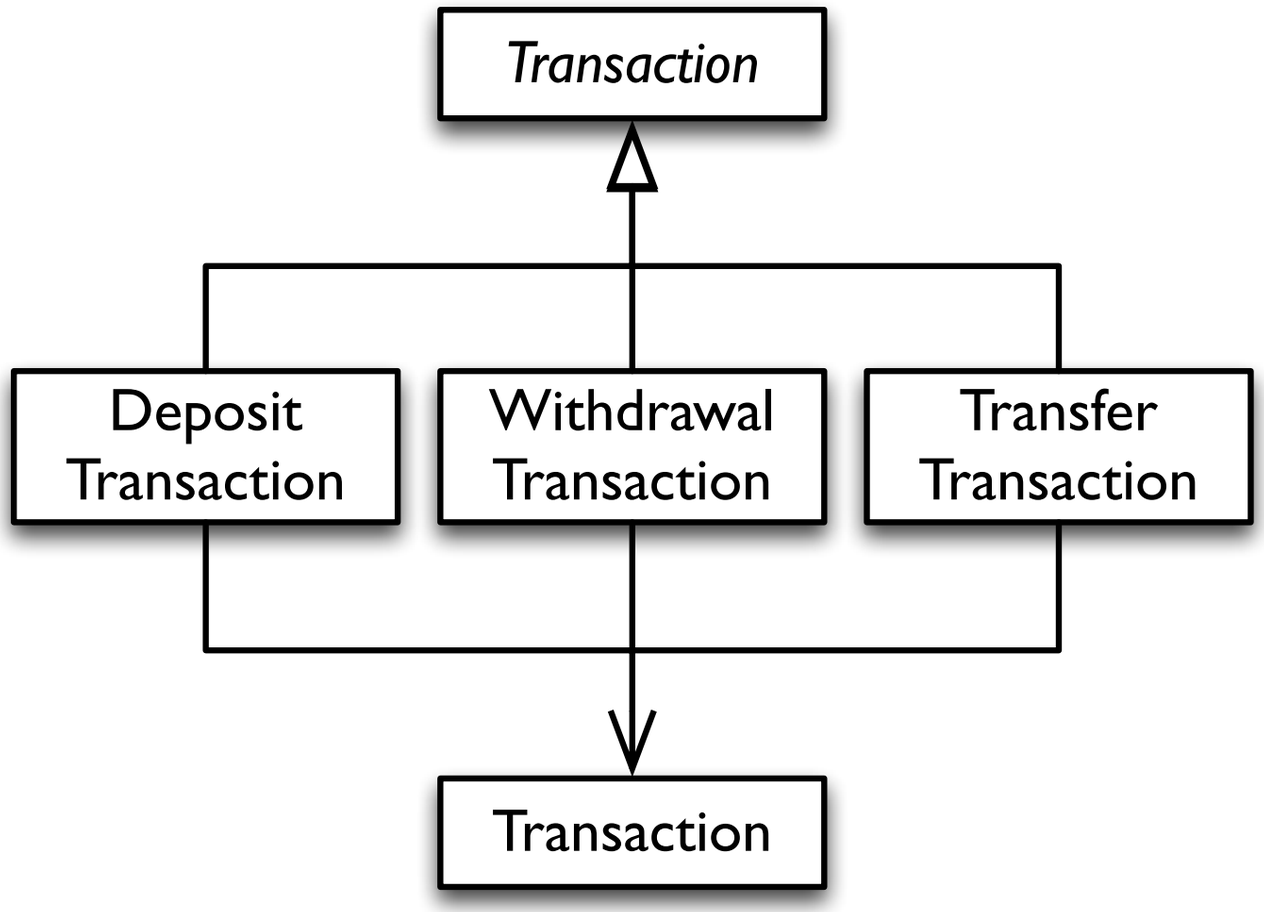
- Derived classes should be usable through the base class interface, without the need for the user to know the difference.

Liskov substitution principle



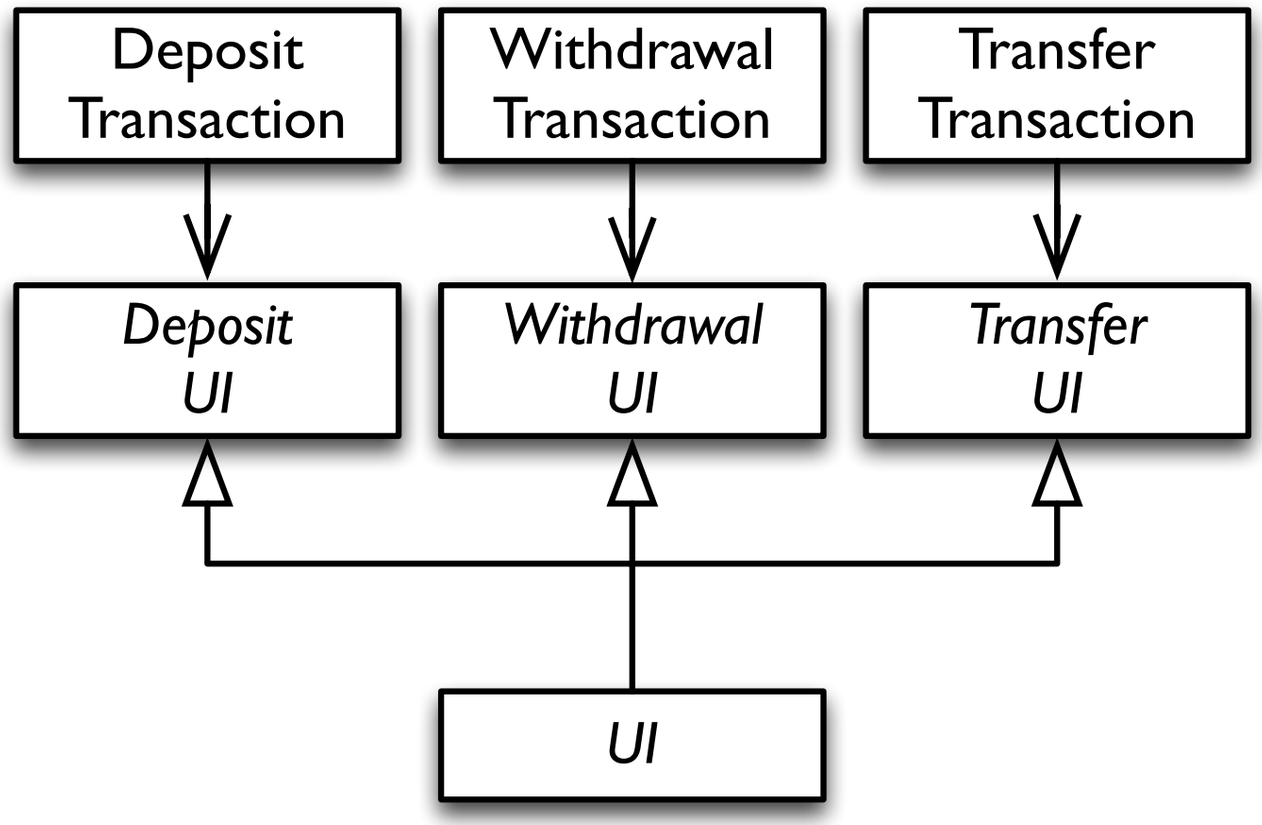
- The inheritance relationship is just the redeclaration of functions and variables in a sub-scope.
- They do not share the same number of variables, they do not share behaviour, they are not related at least not as siblings.

Interface segregation principle

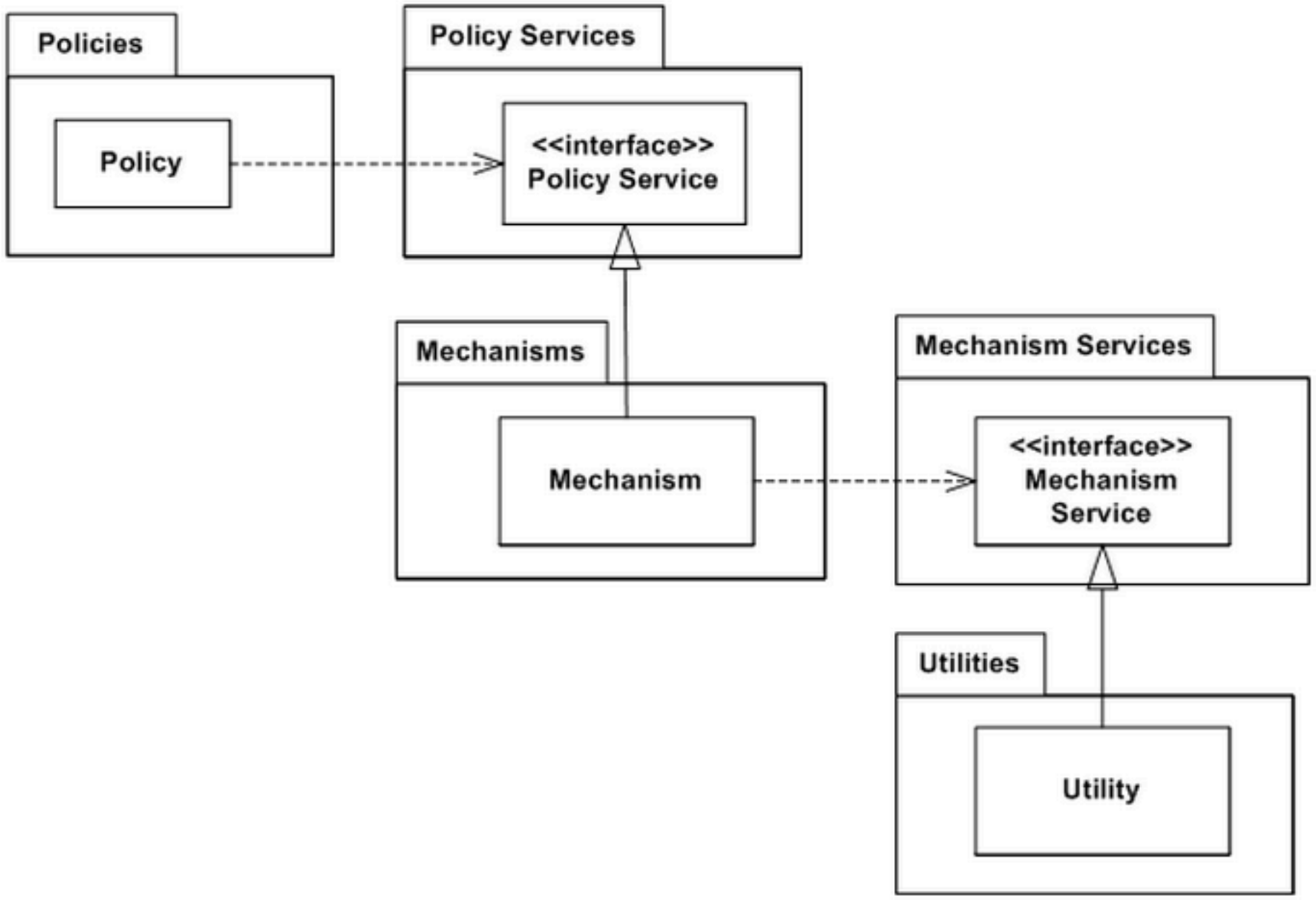


- Many client-specific interfaces are better than one general-purpose interface.

Interface segregation principle



Dependency inversion principle



- Depend upon abstractions, do not depend upon concretions.

Concluding

Key points

- Single responsibility principle: “a class should only have one, and only one, reason to change”.
- Open/closed principle: “software entities should be open for extensions but closed for modifications”. (Bertrand Meyer 1988)
- Liskov substitution principle: “derived classes should be usable through the base class interface, without the need for the user to know the difference”. (Barbara Liskov 1987)
- Interface segregation principle: “many client-specific interfaces are better than one general-purpose interface”.
- Dependency inversion principle: “depend upon abstractions, do not depend upon concretions”.

Outline

- Literature
  - [OOSE] ch. 6
  - [SE9] ch. 6

- Topics covered:
  - SOLID principles