Functional testing

Barbara Russo

SwSE - Software and Systems Engineering

Functional testing (Black Box testing)

- Deriving test cases from **program specifications** (e.g. selecting inputs and oracles)
- Functional testing does not exploit design or code (white-box testing)
- Functional testing is the baseline technique for any other testing strategy
- **It is independent from any implementation (design or code)**

Basic approach

- Why not simply picking random input to design test cases?

Random testing

- *Picking inputs according to a uniform distribution*
- 👍It avoids designer bias
  - The test designer can make the same logical mistakes and bad assumptions as the program designer
- 👍It limits costs, it does not require much knowledge of the input
- 👍It can be automatized and produce more test cases than partition testing

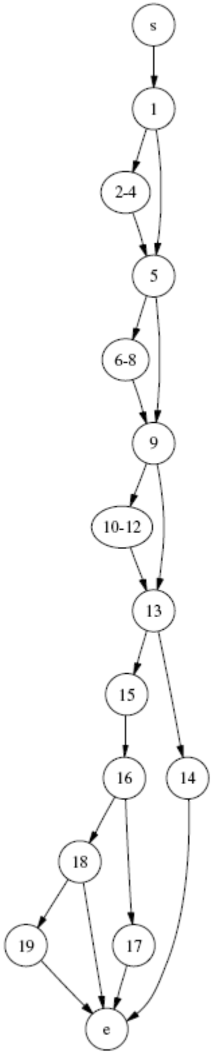
Random testing

- 👎It treats all inputs as equally valuable
- 👎It is not able to pick specific / critical input values as it treats all inputs the same

- Random testing: execute the program with random inputs and observe the code coverage

- Weakness: structures having a low probability of being executed are often not covered

```
s      int tri_type(int a, int b, int c)
       {
           int type;

1          if (a > b)
2-4        {   int t = a; a = b; b = t;    }

5          if (a > c)
6-8        {   int t = a; a = c; c = t;    }

9          if (b > c)
10-12      {   int t = b; b = c; c = t;    }

13         if (a + b <= c)
           {
14             type = NOT_A_TRIANGLE;
           }
           else
           {
15             type = SCALENE;
16             if (a == b && b == c)
               {
17                 type = EQUILATERAL;
               }
18             else if (a == b || b == c)
               {
19                 type = ISOSCELES;
               }
           }

e          return type;
       }
```



Exercise

- Discuss random testing for the following code.
- How can it discover the bug?

Exercise

```
public class Pair {
            public Pair(){}
  public Double x;
  public Double y;


}
```

```
public class SquareRoot {

    public  Pair solve(double a, double b, double c){
        Pair myPair = new Pair();
        double q= b*b-4*a*c;
        System.out.println("The value of q is "+q);
        if (a!=0 && q>0){
            myPair.x = (0-b+Math.sqrt(q))/2*a;
            myPair.y =(0-b-Math.sqrt(q))/2*a;
        } else if (q==0){ // Bug
            myPair.x =(0-b)/(2*a);
            myPair.y =(0-b)/(2*a);
            }
        else {System.out.println("The solutions are imaginary numbers");}
        System.out.println("The solutions are "+ myPair.x +" and "+ myPair.y);
        return myPair;

    }

    public static void main(String[] args){
        SquareRoot mySR=new SquareRoot();
        mySR.solve(Double.parseDouble(args[0]), Double.parseDouble(args[1]),
Double.parseDouble(args[2]));
    }
}
```

Discussion

- Random test case generation is fine to test for $q > 0$
- Random sampling unlikely picks a=0.0 and b=0.0

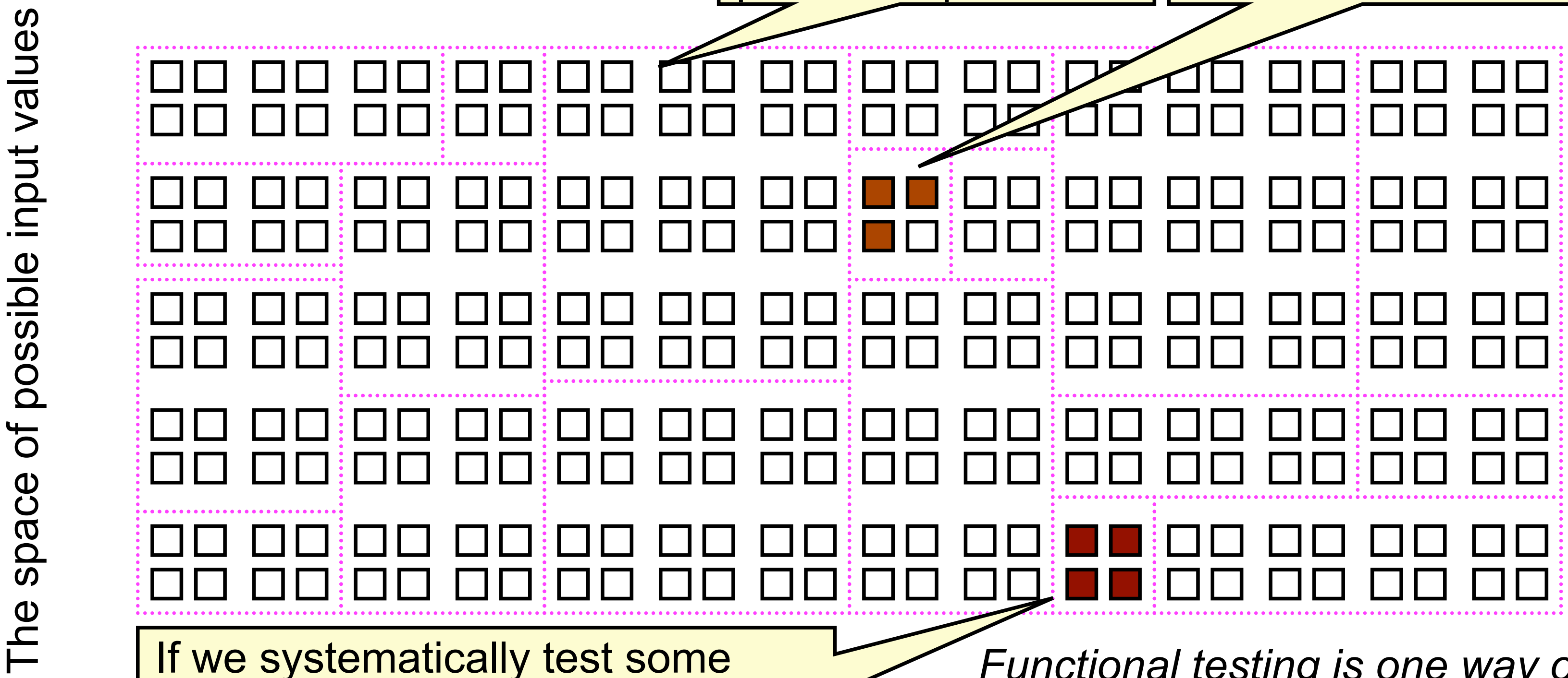
Systematic Partition Testing

🟥 Failure (valuable test case)

☐ No failure

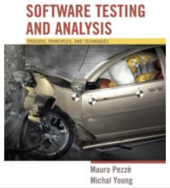
Failures are sparse in the space of possible inputs ...

... but dense in some parts of the space



If we systematically test some cases from each part, we will include the dense parts

Functional testing is one way of drawing pink lines to isolate regions with likely failures



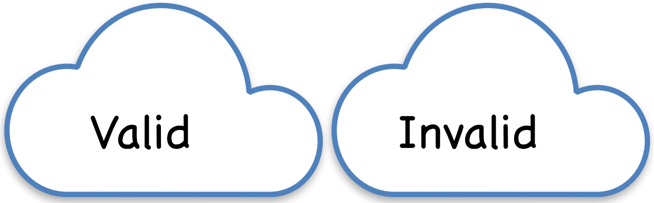
Functional testing uses partition and boundary

- Functional testing uses the specification (formal or informal) to partition the input space
  - E.g., the specification of “roots” program suggests division between cases with zero, one, and two real roots
- **Test each part, and boundaries between parts**
  - No guarantees, but experience suggests failures often lie at the boundaries

The partition principle

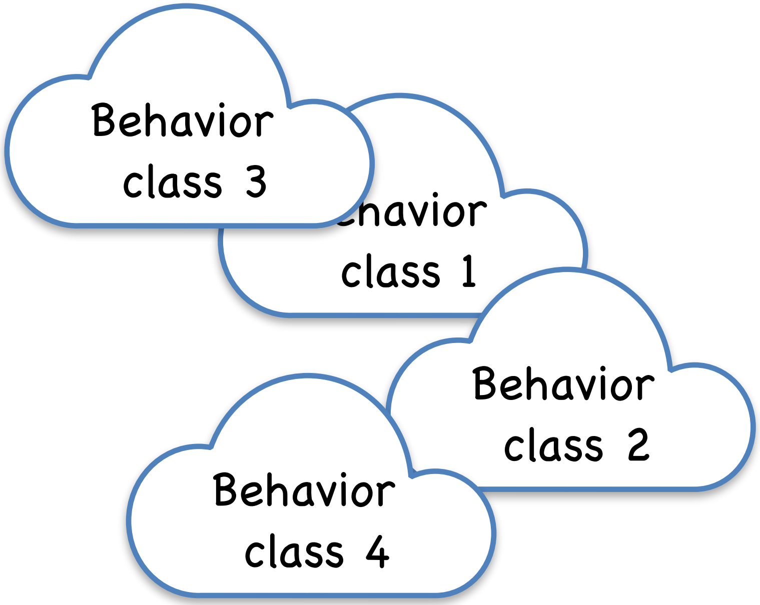
- In principle, it divides (infinite) input into a finite number of classes where each class can be homogeneously associated to **one output success** or **failure**
- Partition divides input into a finite set of classes of program behaviour
- For example y=abs(x): Class1=X>=0 Class2=X<=0

Check point! Partition



Valid and invalid input refers to the primary goal of the functionality described in the specifications; careful: invalid does not mean failure

Failure and success concern testing the specific implementation; classes can change with system's state and the environment's changes



behavior can be non deterministic; it can change with the system's state and the environment's changes; careful: classes can overlap

Partition- selecting representatives

- Tests are designed on **representatives (input)** of classes
- Often classes and representatives are defined by using expert opinion

Partition- selecting representatives

- *We do not know which testing strategy would be likelier to reveal faults*:
  - Repeating the **same/similar test case** is **less likelier** to find a fault than exercising a **different test case**

Partition- selecting representatives

This is a specification.

- What is the input to consider?
- Which are the classes of behavior?
- What is valid or invalid?
- What is success or failure?

Partition- selecting representatives

- Example: split a buffer into lines of length 60 characters

This is a specification.

- What is the input to consider?
- Which are the classes of behavior?
- What is valid or invalid?
- What is success or failure?

Partition- selecting representatives

- Example: split a buffer into lines of length 60 characters
  - **Just four test cases are available:** Buffer of length 16, 30, 40 and 100. Which test case is more valuable?

This is a specification.

- What is the input to consider?
- Which are the classes of behavior?
- What is valid or invalid?
- What is success or failure?

Partition- selecting representatives

- **Random generation of test cases** with uniform distribution would avoid this specific distribution of test cases
  - but it would be likelier to find faults in buffers with lengths greater than 60 (higher cumulative probability)

The partition principle

- 👎Limitation: selecting representatives might be expensive
- 👍More efficient on particular regions where fault are dense, but
    - 👎Localising dense faulty input areas requires expert judgment or advanced techniques of **search based testing**

Boundary testing

- 👍Boundary testing exercises values **on the boundary of classes**
- It requires thorough knowledge of input, often it needs manual investigation
- 👎Limitation: Expensive

Brute force testing

- In the example, specifications were simple, but
- 👎*Direct generation of test cases from specifications (***brute force***) might be complex* and produces unacceptable results
- There is a need of a systematic general procedure

Systematic functional testing

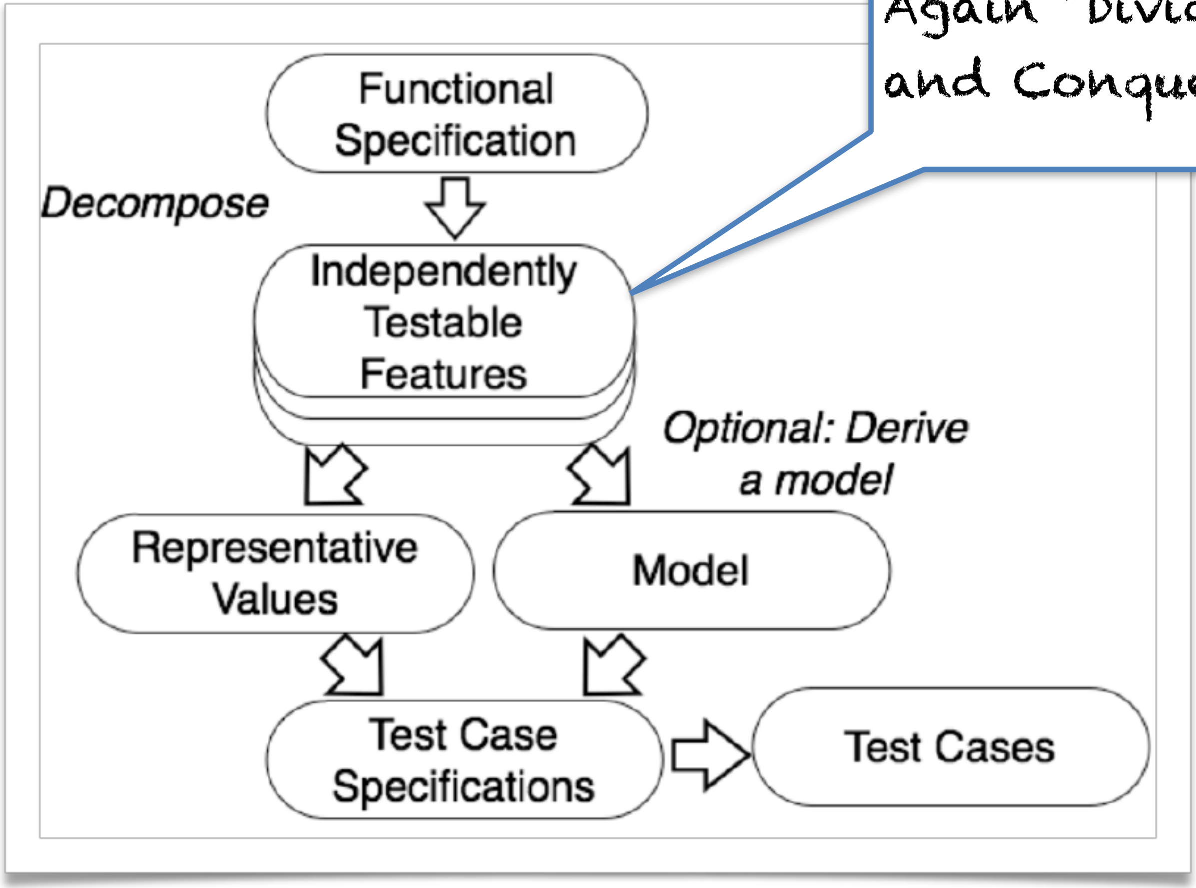
Barbara Russo

SwSE - Software and Systems Engineering

Systematic Testing

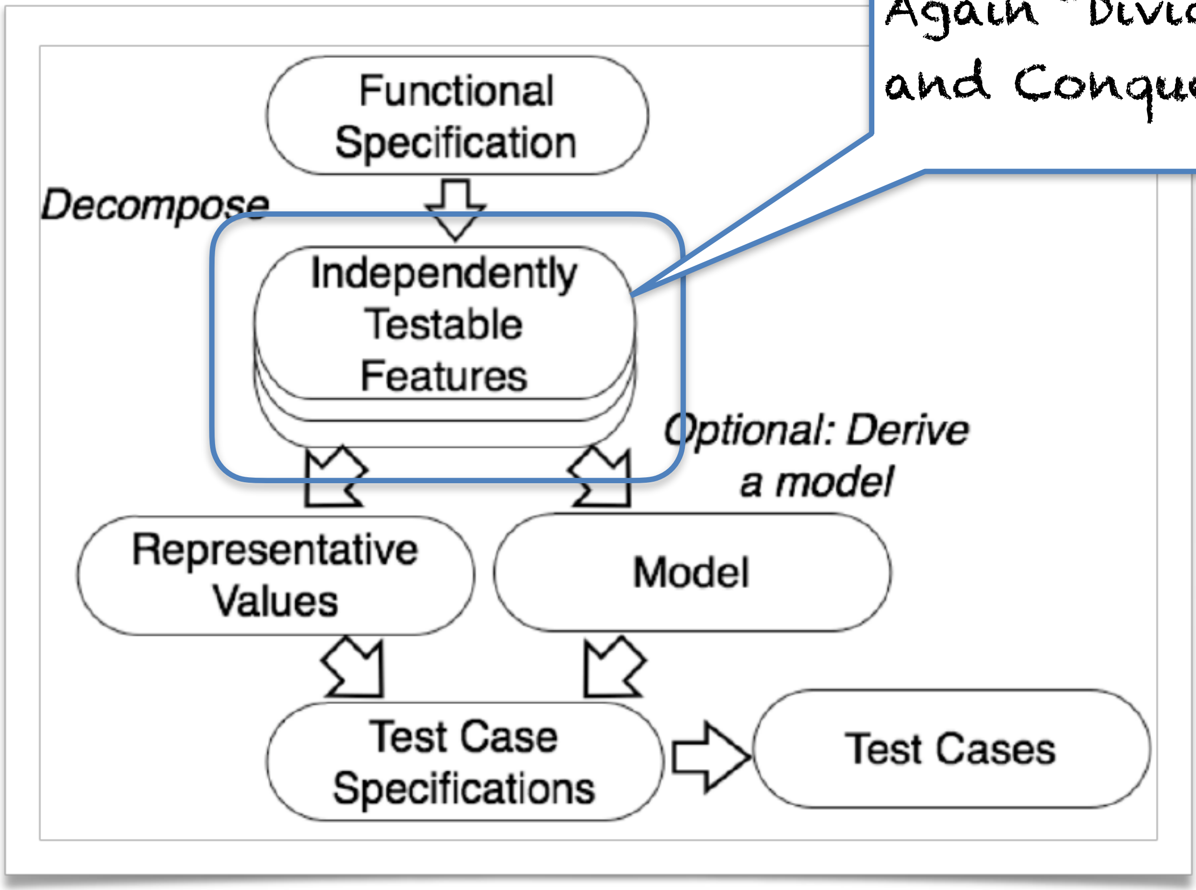
- Systematic (non-uniform):
  - Try to select **inputs** that are **especially valuable**
  - *Usually by choosing representatives of classes that are apt to fail often or not at all*

Steps in systematic functional testing



Again "Divide and Conquer"

Steps in systematic functional testing

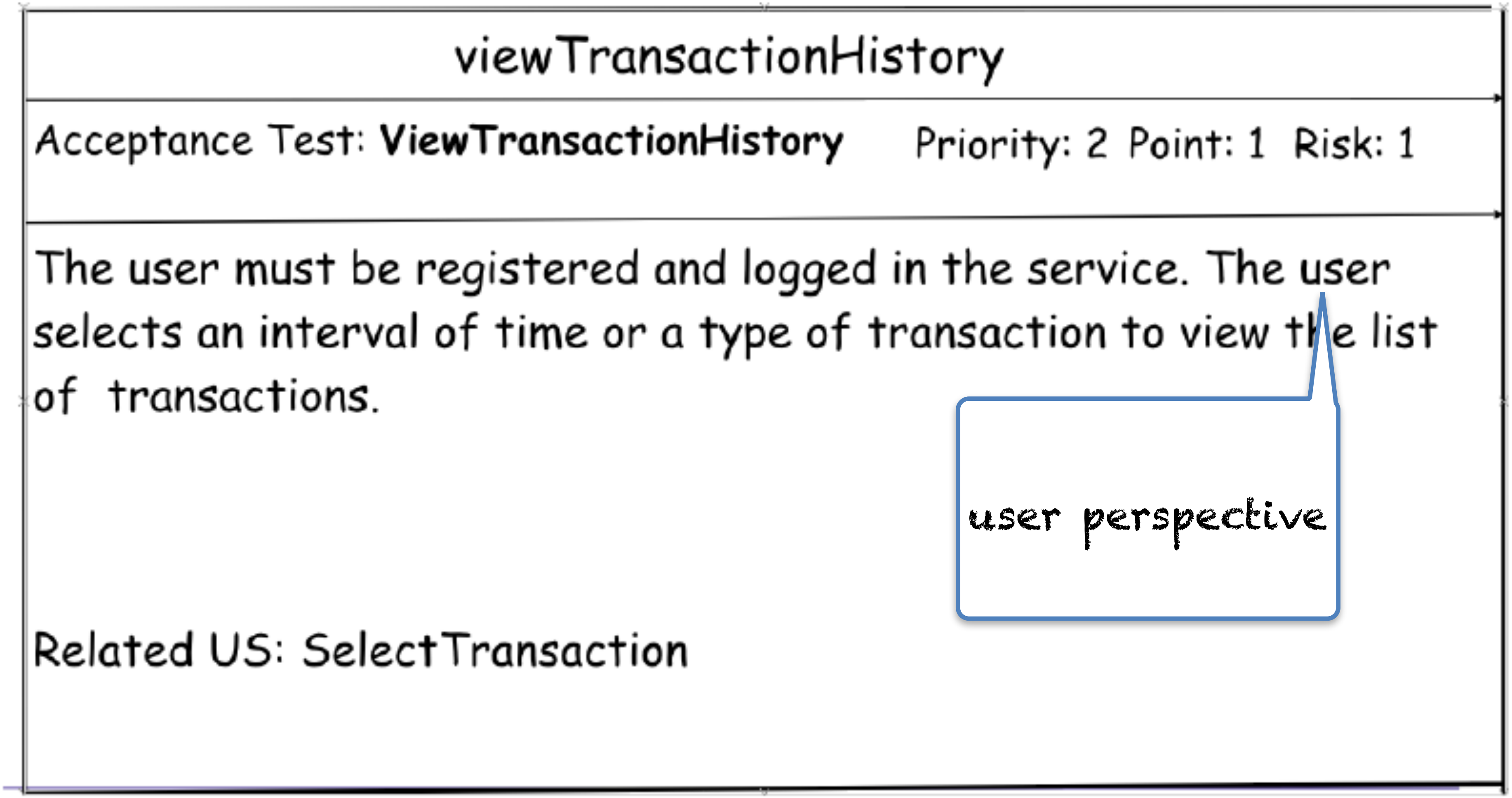


Again "Divide and Conquer"

Identify independent testable features

- Goal: *partitioning specifications into features that can be tested separately*
  - How: Divide features by **functional use** as perceived by **users**

User Story



viewTransactionHistory

Acceptance Test: **ViewTransactionHistory** Priority: 2 Point: 1 Risk: 1

The user must be registered and logged in the service. The user selects an interval of time or a type of transaction to view the list of transactions.

Related US: SelectTransaction

user perspective

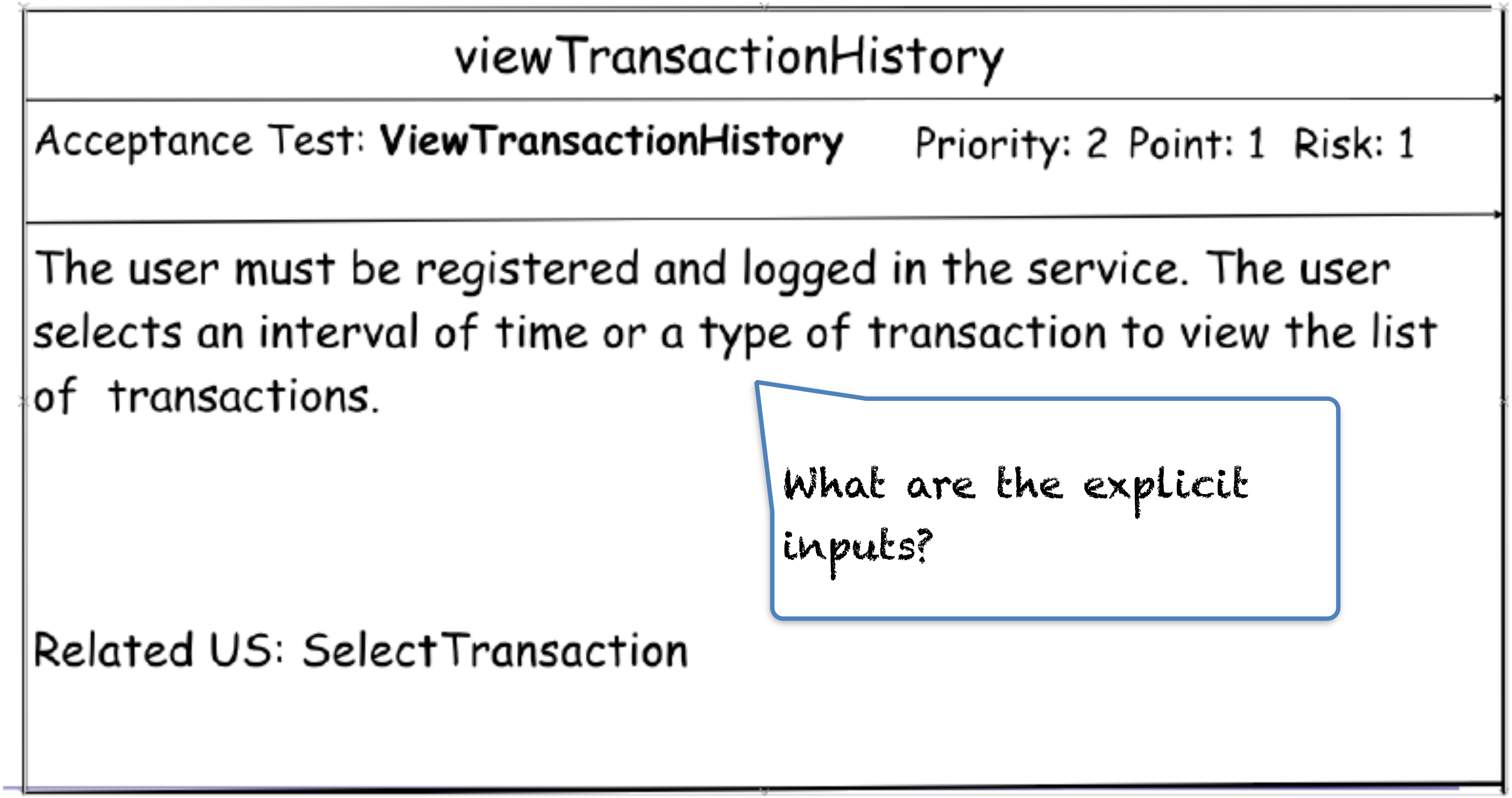
How to detect features?

- Features are identified by all the inputs that determine the **execution behavior**
- These inputs can have different forms, they can be **explicit or implicit** in the **specifications** or inputs for **some program model** (e.g., inputs that trigger the states in the finite state machine) that describes the system behavior

Independent features (XP)

- Identify independent features
  - From **User Stories** (XP), identify implicit and explicit input
  - From the **system metaphor (XP),** identify **implicit** form of input to augment the explicit definition in the user stories

User Story



viewTransactionHistory

Acceptance Test: **ViewTransactionHistory** Priority: 2 Point: 1 Risk: 1

The user must be registered and logged in the service. The user selects an interval of time or a type of transaction to view the list of transactions.

What are the explicit inputs?

Related US: SelectTransaction

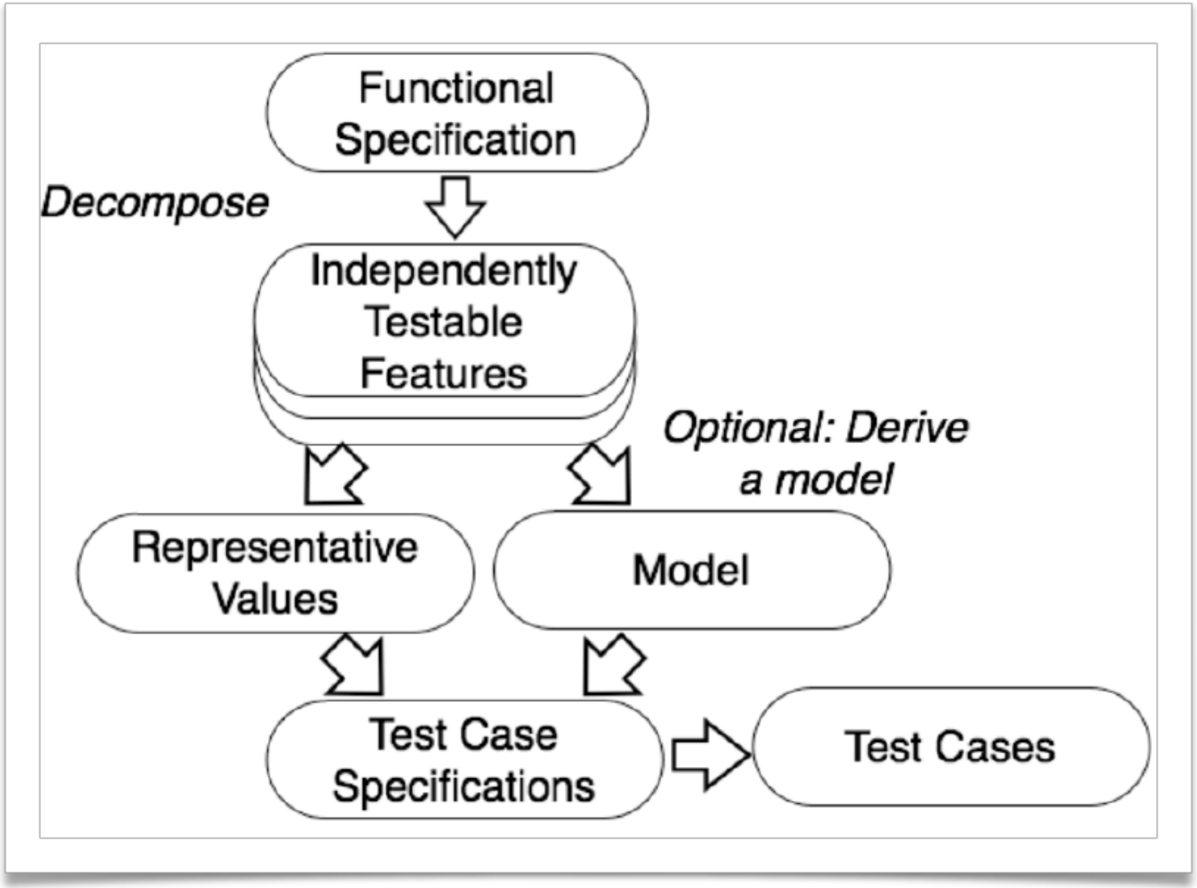
Exercise

- An automatic coffe machine
- What are the explicit input?
- What are the implicit input?

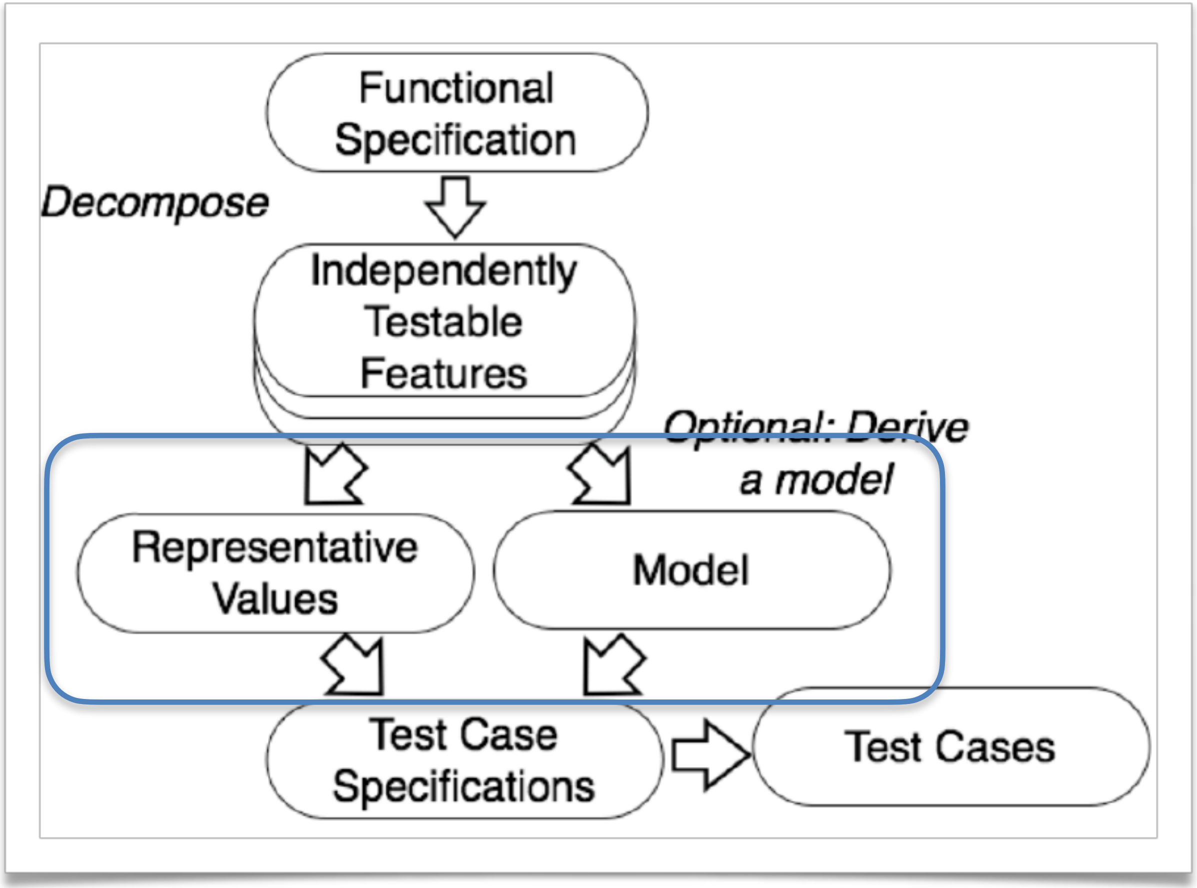
Example of input from a metaphor

- In a coffe machine scenario, the ingredients that are assembled with the water

Steps in systematic functional testing



Steps in systematic functional testing



Select the values of input

- There are two practicable ways:
  - **Representative** values of input (implicit and explicit)
  - Derived from a **model**: e.g., control flow graph or finite state machine

Identify inputs and their characteristics

- **Implicit and explicit** parameters
- Their **elementary characteristics**
- The **environment elements and characteristics** that effect the execution of the feature in a given unit of work (like DBs that are required to execute test cases)
- **Categories of parameters' values** defined by *system behavior* and pick a representative value

Example

Parameter

Characteristics

- Coffe machine **parameters and characteristics**
  - coffee (explicit): amount, temperature, poured
  - sugar (explicit): amount, type, poured
  - powder (implicit): amount
  - temperature (implicit): limit, scale
- Environmental elements
  - card: credit amount

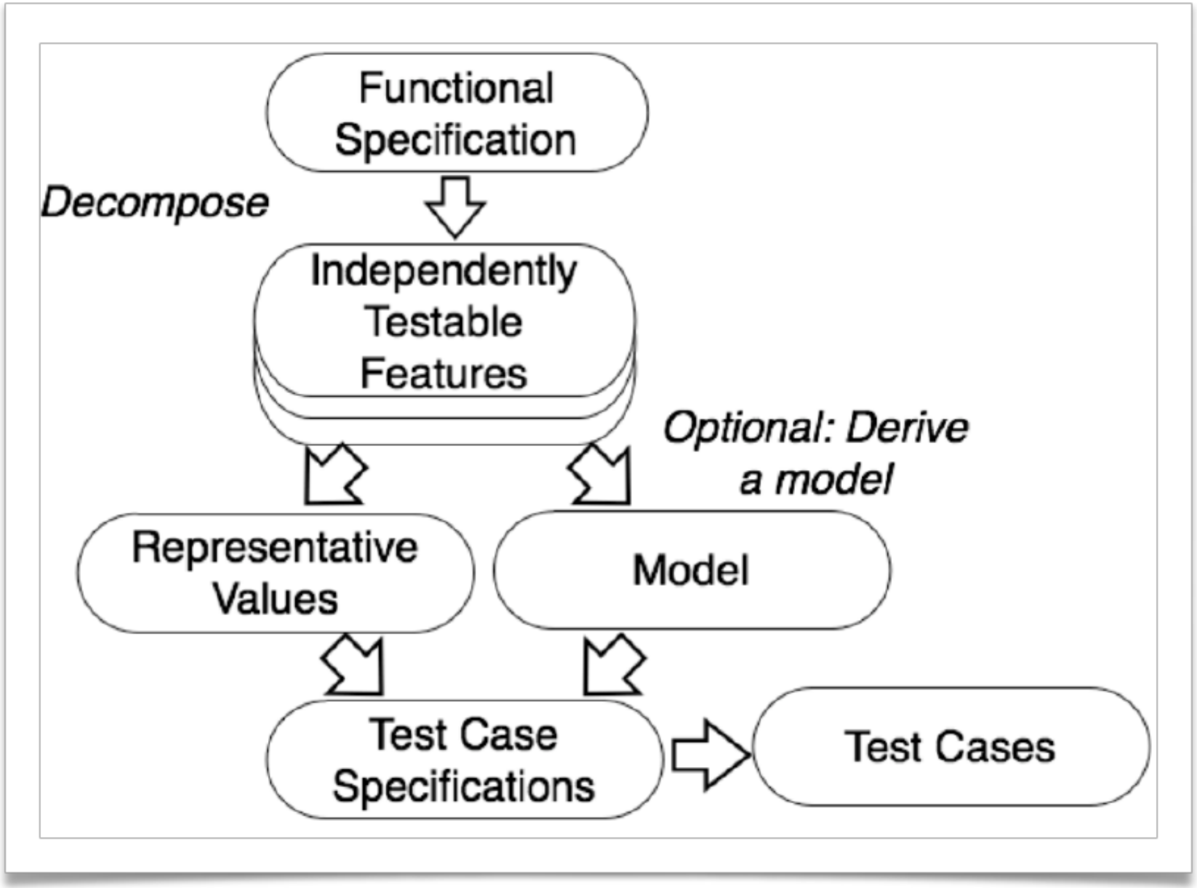
Categories

Categories of values

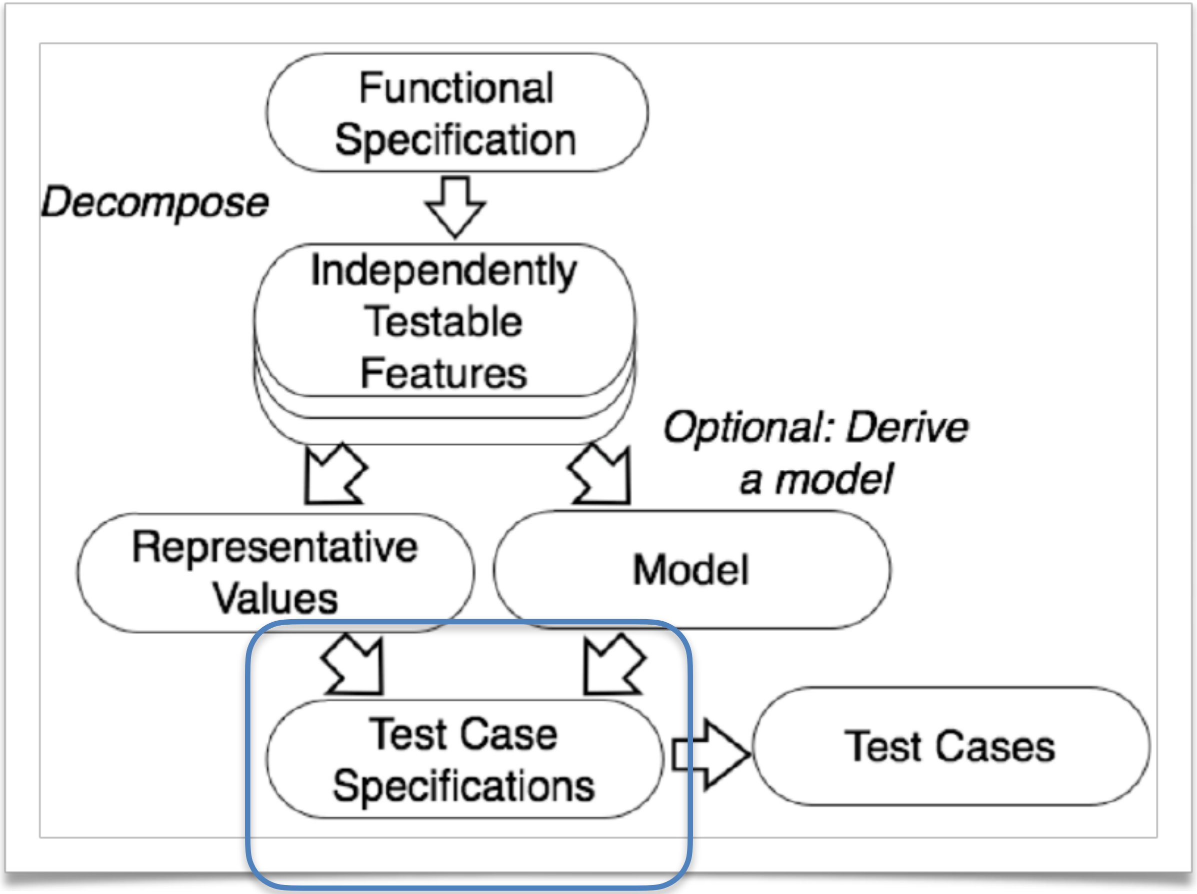
- Coffe machine parameters
  - coffee: amount (categories: 0, positive, # over limit)
  - temperature: scale (categories: F, C)
  - temperature in Celsius: (categories: positive more than default, default, positive less than default, 0)

- Environmental elements
    - card: credit amount (categories: 0, positive less than needed, positive more than needed, needed)

Steps in systematic functional testing



Steps in systematic functional testing



Generate test case specifications

- Test Specifications are built by **combining the input values** (e.g., representative)
- *Brute force combination of values might be very expensive*: 5 input variables with 6 values each produces 6 to the 5 test cases
- *Reducing the inputs space is crucial to reduce the effort of test designing*

Example - acceptance testing

logInTest		
Input	**Description**	**Output (behaviors)**
username= initials of name+surname password= 8 characters, alpha numeric, bounded length environmental input (DB) status= student	Preconditions: the user has accessed to the general site of the university The user introduces username and password Postcondition: the user is logged in	logged in not logged in

Example - acceptance testing

- A combination of the input values of username, password, and status is extracted from the **test case specification**
    - For example “the user (brusso, 123456th, professor) shall not log in”.
- How many combinations of input values?
- We need to trade off between coverage and budget (e.g., testing time)

How to reduce input combination

- Combinatorial testing, examples:
  - *Pairwise combination testing*
  - *Category-partition testing*

Pairwise combination testing

- It generates k-ways combinations (typically $k=2$) of categories with $k < n$: $bin(n,k)$
- 👍It goes blindly and does not require a specific knowledge of the domain:
  - 👎it may be still expensive and not effective on sparse faults

Category-partition testing

- Major characteristics
  - 👍It allows test designers to add constraints and limit the number of test cases
  - 👍Useful **when** we have enough knowledge of the domain and its constraints (e.g, what is valid and what is not)
  - 👍It works with all kind of data structures

Category-partition testing

- Major characteristics
  - **Flatten data structures** into **parameter characteristics**
  - **Filter out combinations** of values in the generation of the test case specifications:
    - First, label categories
    - Then, use labels to rule out infeasible combinations

Example - Flattening data

Parameter

- *Computer Model* (data structure) is
  - **ID key**, integer used to search and retrieve from DB, **model number, number of required slots**, and **number of optional slots**

Characteristics

Category-partition testing

- Labels of parameter characteristics:
  - [error],
  - [single],
  - [property: <Acronym>]

- If condition [if <Acronym>]

Category-partition testing

- The **labelling** requires expert judgment, some characteristics might be erroneous only in combination with other characteristics

Category-partition testing

- “**[error]**” : a category needs to be tried in combination with non-error categories of other characteristics **only once**
- “**[single]**” acts as “[error]” but for any type of values (error or not).
  - This is not a real constraint coming from the domain, *it is set by the designer to reduce the number of combinations!*

Category-partition testing

- “**[property:]**” qualifies categories of values
- The **if condition** uses the properties to identify logical constraints between categories
    - These are used to rule out combinations that are not feasible

Example “Check configuration” feature

Feature: *Check the computer configuration against a reference catalogue (DB)*

Identify parameters

- **Parameters**: Model and components
- **Model**: represents a specific product and determines a set of constraints on the available components (like screen, hard disk, processor etc)
- **Component**: a logical slot which might or might not represent a physical slot on a bus

Identify parameters

- **Components**: a collection of <component, selection>
  - A selection is a choice of a physical slot
- **Environmental variable**: Database of models and components that is required to execute the feature

Identify parameter characteristics

- Computer Model:
  - **ID key**, integer used to search and retrieve from DB, **model number, number of required slots**, and **number of optional slots**

Identify parameter characteristics

- Components <component, selection>
  - **number of required / optional slots** with non-empty selection, **compatibility** of selection and the component (e.g., 20 gigabyte of hard disk (*selection*) for the hard disk slot (*component*))
- External environment
  - DB: **number** of models in DB, **number** of components in DB

Categories of values

- "Components"
- We first **flatten** the data structure Components (<component, selection>) to characteristics:
  - Compatibility of selection with **component**
  - Compatibility of selection with **model**
  - Matching selection and **DB entry**
  - Compatibility of selection with **another component**

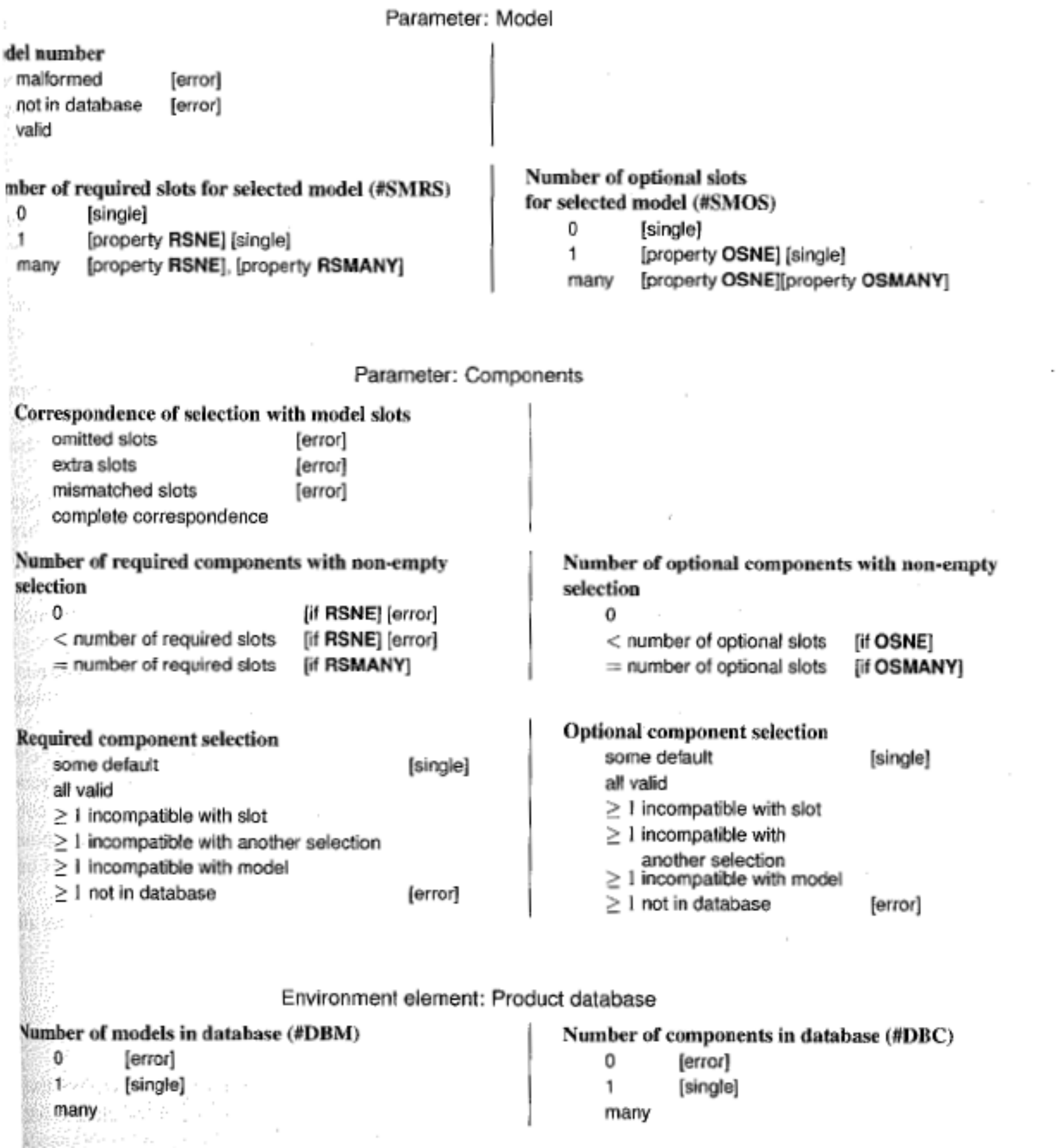
Categories of values

- Then we select a category for *Components*, for example: *Compatibility of selection with the component*
  - We can represent components as an **array of compatible/non-compatible selections**.
  - If array size is n, we have **2^n combinations** of values for the characteristic

C		C		

Categories of values

- Best would be create a test case for all combinations of compatible and non-compatible
  - Often infeasible!
- Simpler value choices: *one compatible, one incompatible, all compatible or all incompatible, selections of slots*
- It is up to the test case designers



Parameter: Model

del number

malformed	[error]
not in database	[error]
valid	

mber of required slots for selected model (#SMRS)

0	[single]
1	[property **RSNE**] [single]
many	[property **RSNE**], [property **RSMANY**]

Number of optional slots for selected model (#SMOS)

0	[single]
1	[property **OSNE**] [single]
many	[property **OSNE**][property **OSMANY**]

Parameter: Components

Correspondence of selection with model slots

omitted slots	[error]
extra slots	[error]
mismatched slots	[error]
complete correspondence	

Number of required components with non-empty selection

0	[if **RSNE**] [error]
< number of required slots	[if **RSNE**] [error]
= number of required slots	[if **RSMANY**]

Number of optional components with non-empty selection

0	
< number of optional slots	[if **OSNE**]
= number of optional slots	[if **OSMANY**]

Required component selection

some default	[single]
all valid	
≥ 1 incompatible with slot	
≥ 1 incompatible with another selection	
≥ 1 incompatible with model	
≥ 1 not in database	[error]

Optional component selection

some default	[single]
all valid	
≥ 1 incompatible with slot	
≥ 1 incompatible with another selection	
≥ 1 incompatible with model	
≥ 1 not in database	[error]

Environment element: Product database

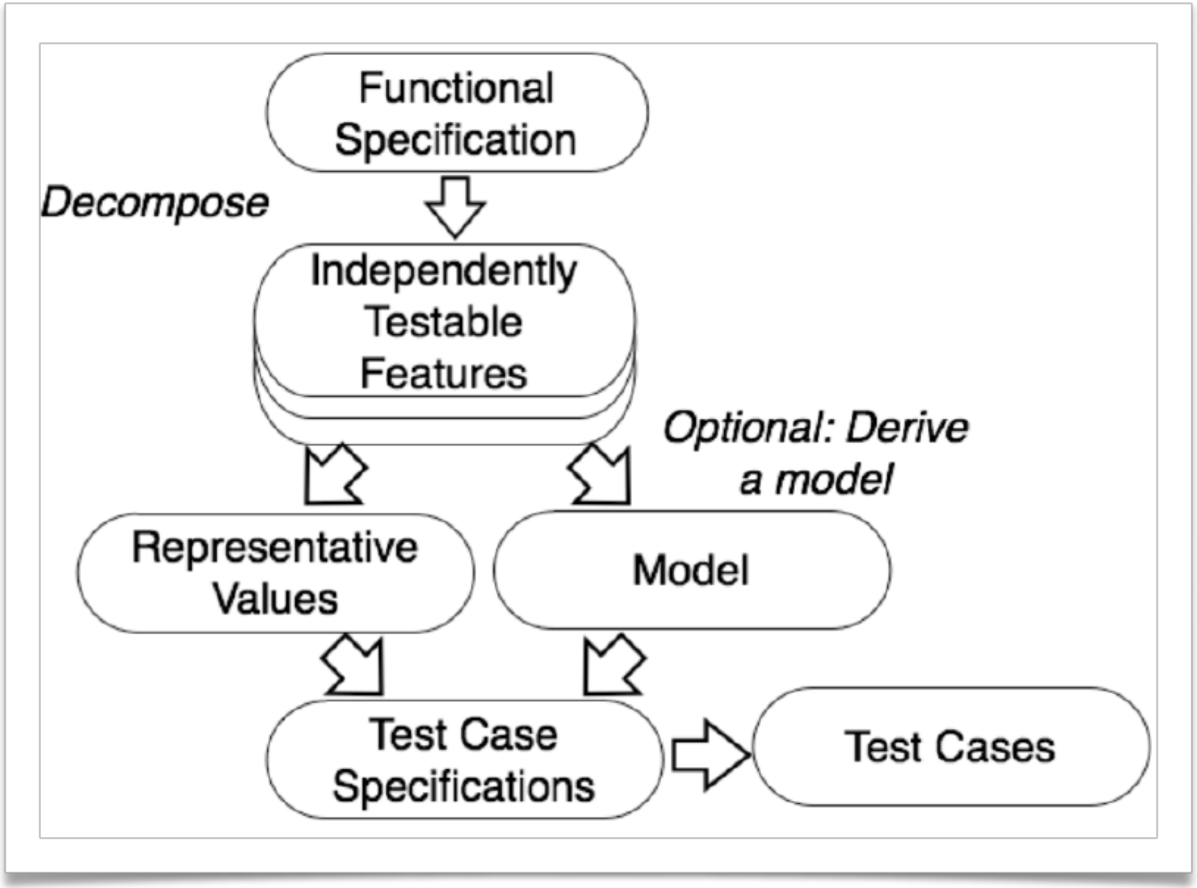
Number of models in database (#DBM)

0	[error]
1	[single]
many	

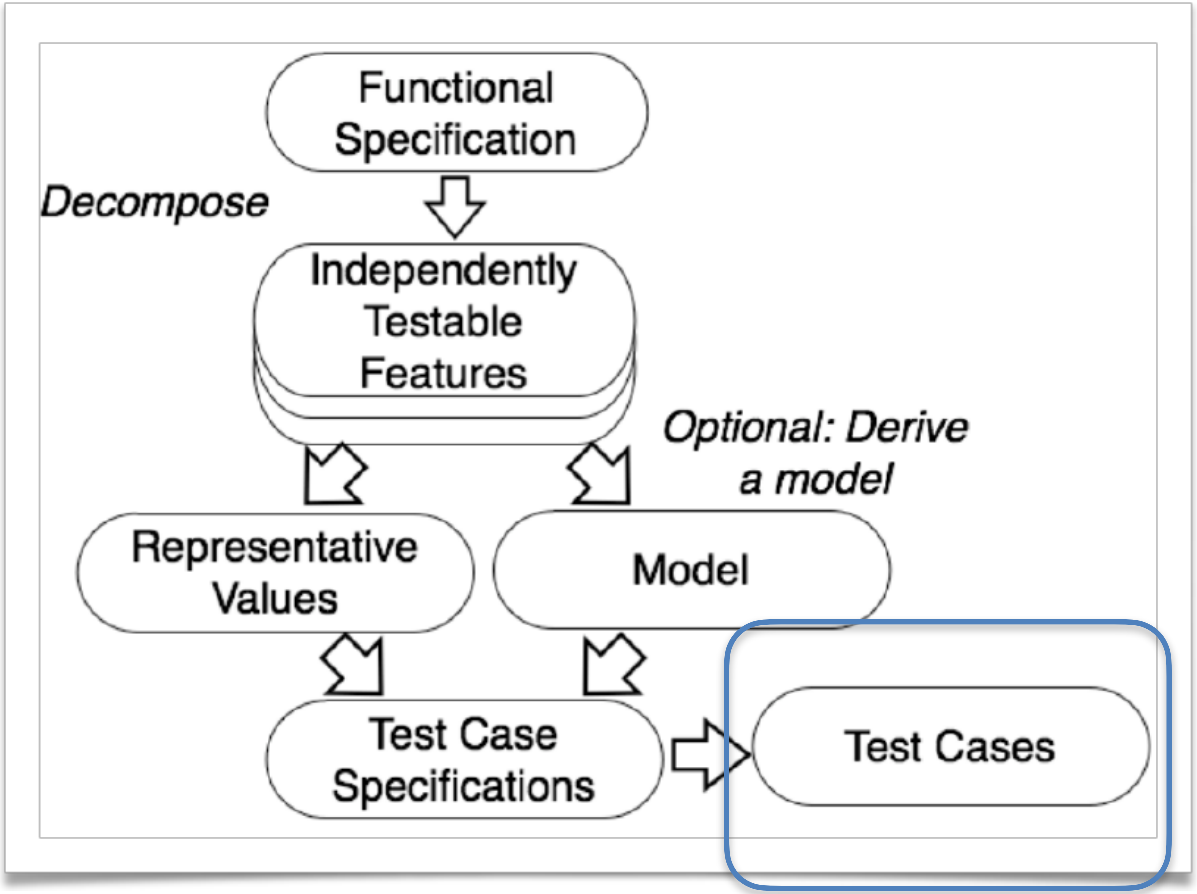
Number of components in database (#DBC)

0	[error]
1	[single]
many	

Steps in systematic functional testing



Steps in systematic functional testing



Generate test case and instantiate tests

- Turning test case specs into test cases
- Implement test cases by defining the harness to execute them (e.g., FitNess)

Exercise

- 05.TestCaseDEsignExercise3
  - inject up to 3 bugs (15’)
  - pass your changed code to the other group
  - design TC that reveal the bugs (15’)
- 06.FunctionalTestingExercise1. For the feature:
  - Define an Adequacy Criterion and
  - Define a TC specification and three obligations (20’)
  - Design three TCs using category partition testing (15’)
  -

Exercise - 10’ presentation

- Read paper SBSTMcMinn
- Present the overall examples
- For the two examples reported define a Feature (one sentence), Test Goal, Test Obligation for which use the metrics proposed in the paper
- 5 slides

- Wednesday 07.04