Qualitative Analysis

Exploring data to create theories

Willian Oizumi - oizumi.willian@gmail.com

Adapted from the material of Leonardo da Silva Sousa and Alessandro Garcia

LES | DI | PUC-Rio - Brasil

OPUS Research Group

Summary

A. Quantitative and Qualitative Analysis

B. Qualitative Analysis

C. Grounded Theory (GT)

D. Theory Representation

E. Identifying Design Problems in the Source Code

F. Concluding Remarks

Quantitative and Qualitative Analysis

Quantitative Analysis

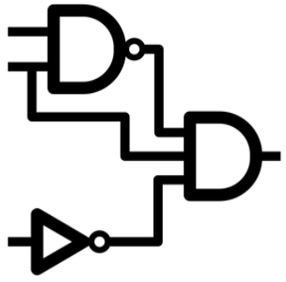
It usually requires the use of statistical methods to reach conclusions

It tends to be objective and without room for interpretations

It is mostly applied for verifying hypotheses

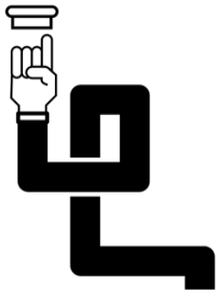
Qualitative Analysis

It often corresponds to an intuitive procedure



It is based on inference

- what does the absence or presence of a given element mean?



It is most malleable at unanticipated events or the evolution of hypotheses

Qualitative Analysis

Qualitative Analysis

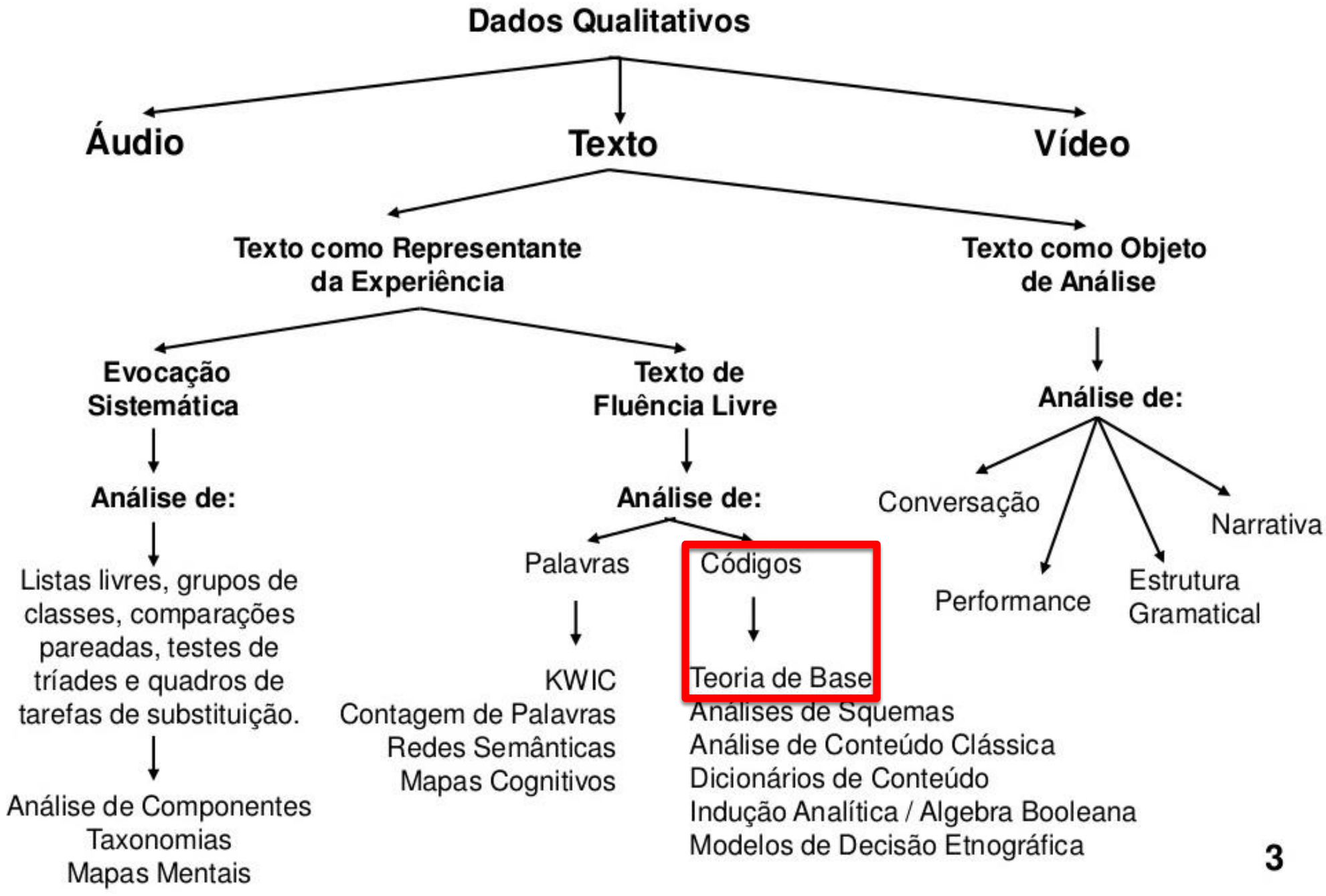
- A process of examining and interpreting data[2] in order to:
  - Elicit meaning
  - Gain understanding
  - Develop empirical knowledge

2. A. Strauss and J.M. Corbin. 1998. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*

Qualitative Analysis

- It allows researchers to:
  - Get at the inner experience of participants
  - Determine how meanings are formed through and in culture
  - Discover and explain rather than test hypothesis

Several Methods to Conduct the Analysis



3. Ryan, G. W. & Bernard, H. R. 2000. *Data management and analysis methods.*

Grounded Theory (GT)

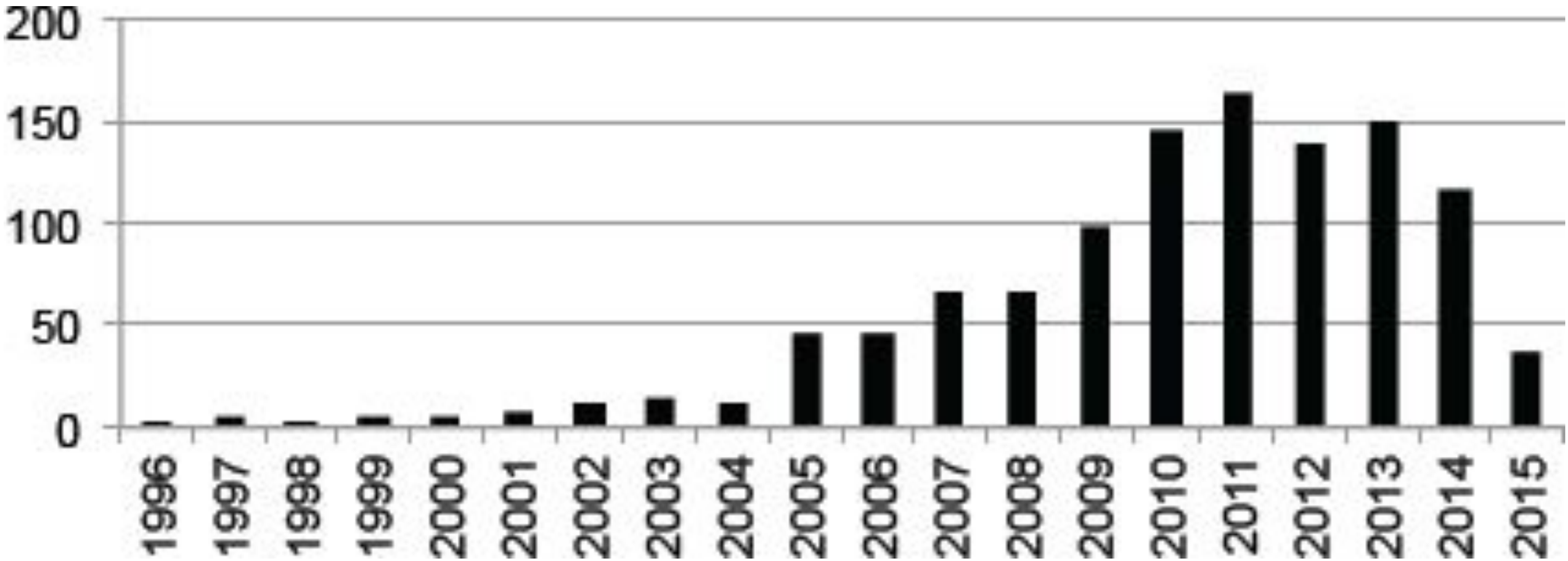
Grounded Theory is a qualitative research method that uses a systematical set of procedures to develop an inductively derived theory about a phenomenon from data[4]

It generates a general explanation of a process, action or interaction

4. A. Strauss and J.M. Corbin. 1998. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*

Grounded Theory in SE

- Rise of grounded theory studies in computer science



Source: Scopus (Aug 2015); search string: TITLE-ABS-KEY ("grounded theory"), limited to "computer science"

Versions of Grounded Theory (1/2)

What constitutes a grounded theory has been labeled as a ‘contested concept’

It is now widely acknowledged that there are at least **three** main versions of GT[5]

Consistency with a particular version is important

5. Adolph, S., Hall, W. and Kruchten, P. 2011. *Using grounded theory to study the experience of software development*

Versions of Grounded Theory (2/2)

Glaser's GT (classic or Glaserian GT)

- strong focus on emergence (of research questions, of codes, of theory)

Strauss and Corbin's GT (Straussian GT)

- meticulous set of 'mini-steps'
- still evolving
- "*more free-wheeling flights of imagination*"

Charmaz's constructivist GT

- resulting theory depend on the researcher's view

Strauss and Corbin's GT

- Strauss and Corbin *go beyond the data* by asking various questions on *what might be* to develop the emerging theory

- Asking questions about **whom**, **when**, **where**, **how**, **with what consequences**, and under **what conditions** phenomena occur, helps to 'discover' important ideas for the theory[6]

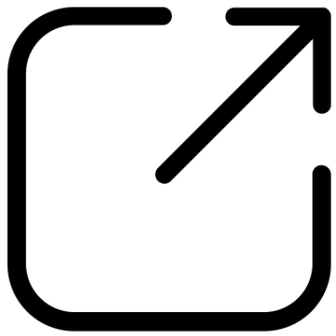
6. Strauss, A. and Corbin, J. 1994. *Grounded Theory Methodology: An Overview.*

Data Collection

- Data can be collected by interview, observation, records, a combination of them, and others
- It results in large amounts of:
  - hand-written notes
  - typed interview transcripts
  - video/audio taped conversations
- Which contain multiple pieces of data to be sorted and analyzed

GT Procedures

- GT comprises of three procedures:
  - Open Coding (1st procedure)
  - Axial Coding (2nd procedure)
  - Selective Coding (3rd procedure)



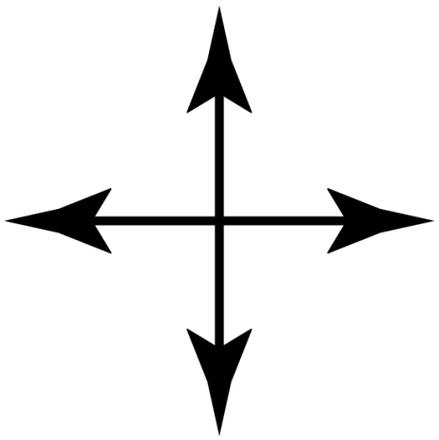
Open Coding involves the breakdown, analysis, comparison, conceptualization, and categorization of the data

Open Coding

- Data are deconstructed into the simplest form possible, examined for commonalities and sorted into categories

“Primeiramente, vamos passar por todas as classes que possam ter uma determinada anomalia”

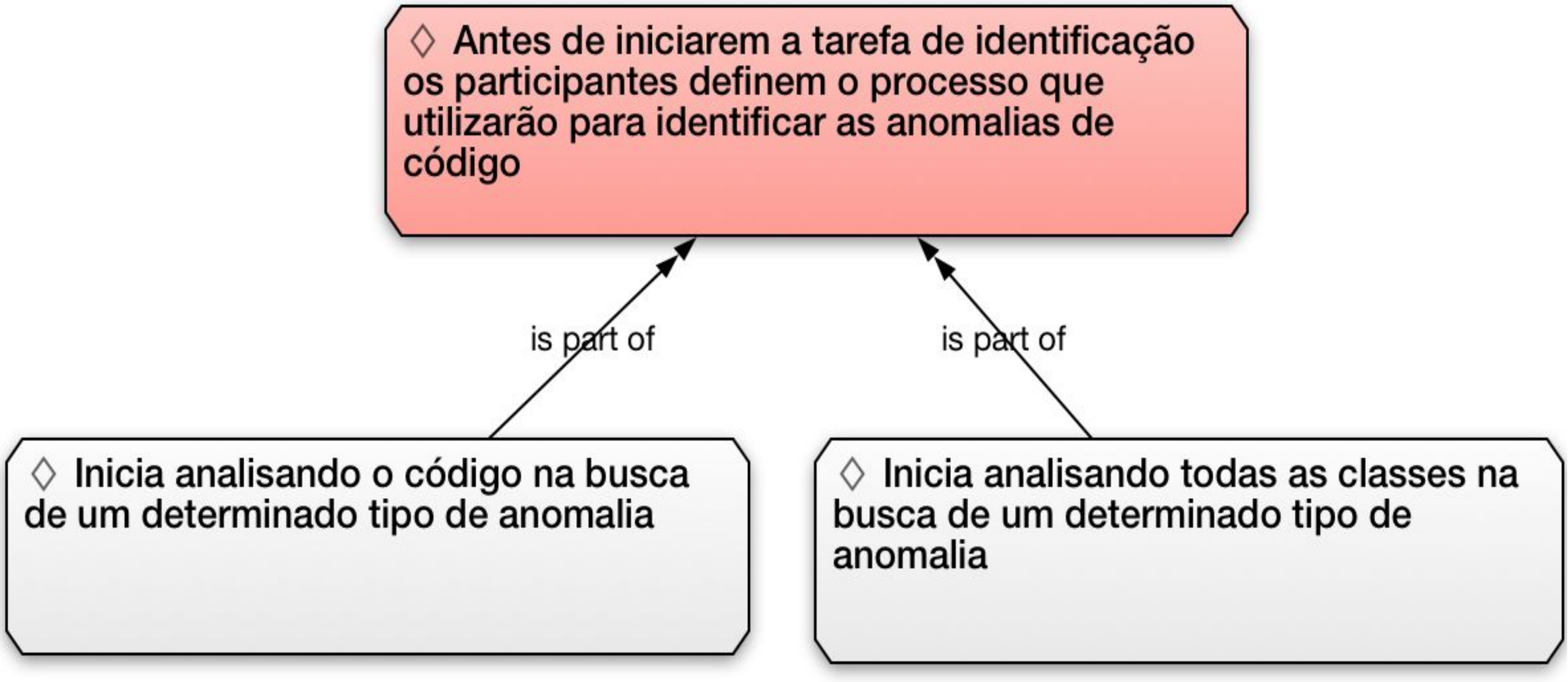
Antes de iniciarem a tarefa de identificação os participantes definem o processo que utilizarão para identificar as anomalias de código



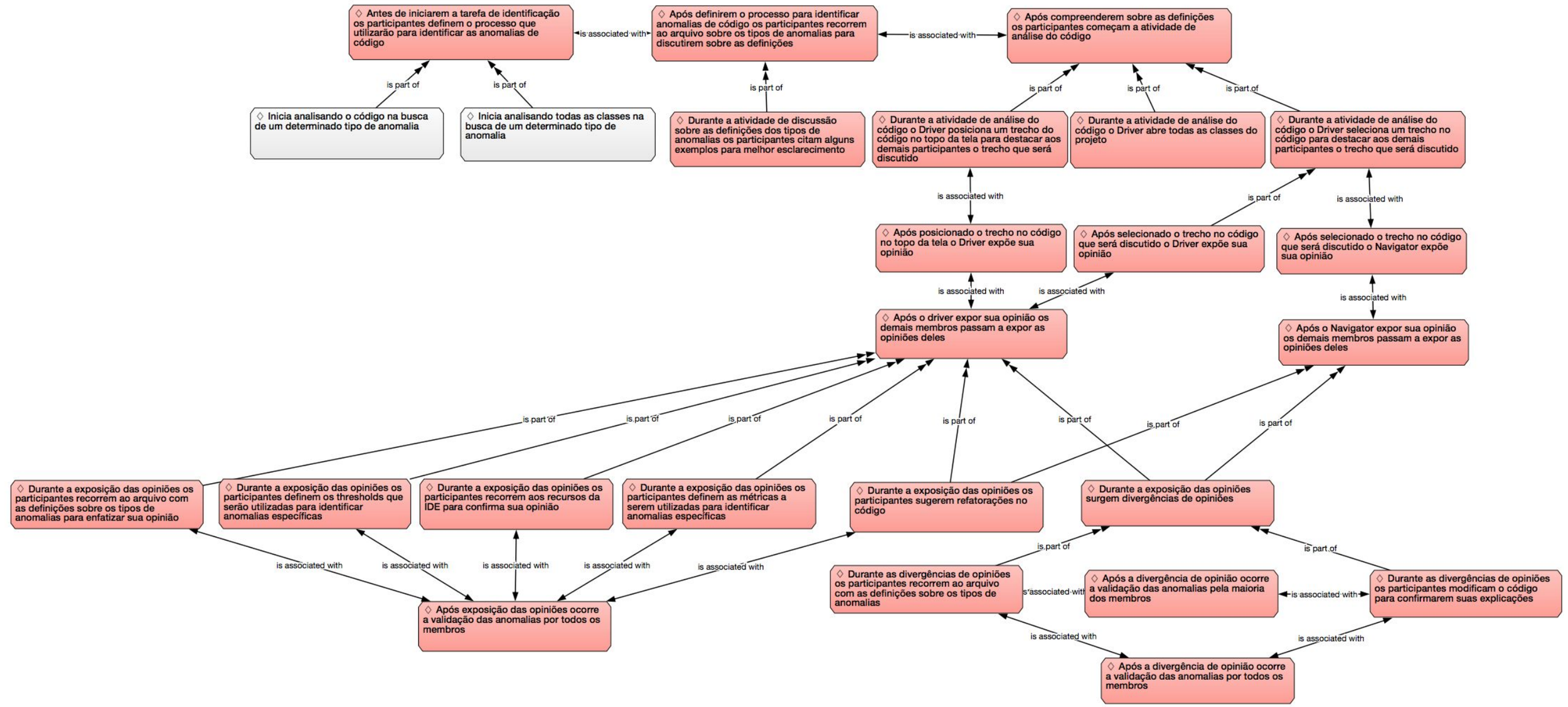
Axial Coding consists in examining the identified categories to establish conceptual relations between them

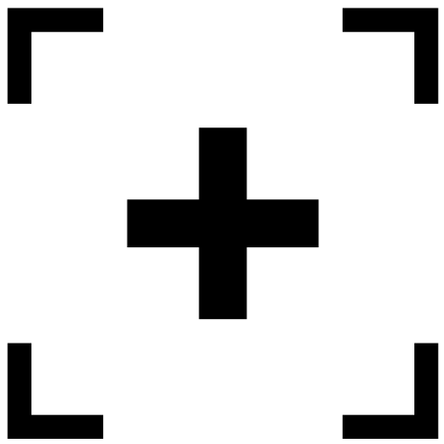
Axial Coding

- Data are reassembled based on logical connections between categories



Graphic Notation



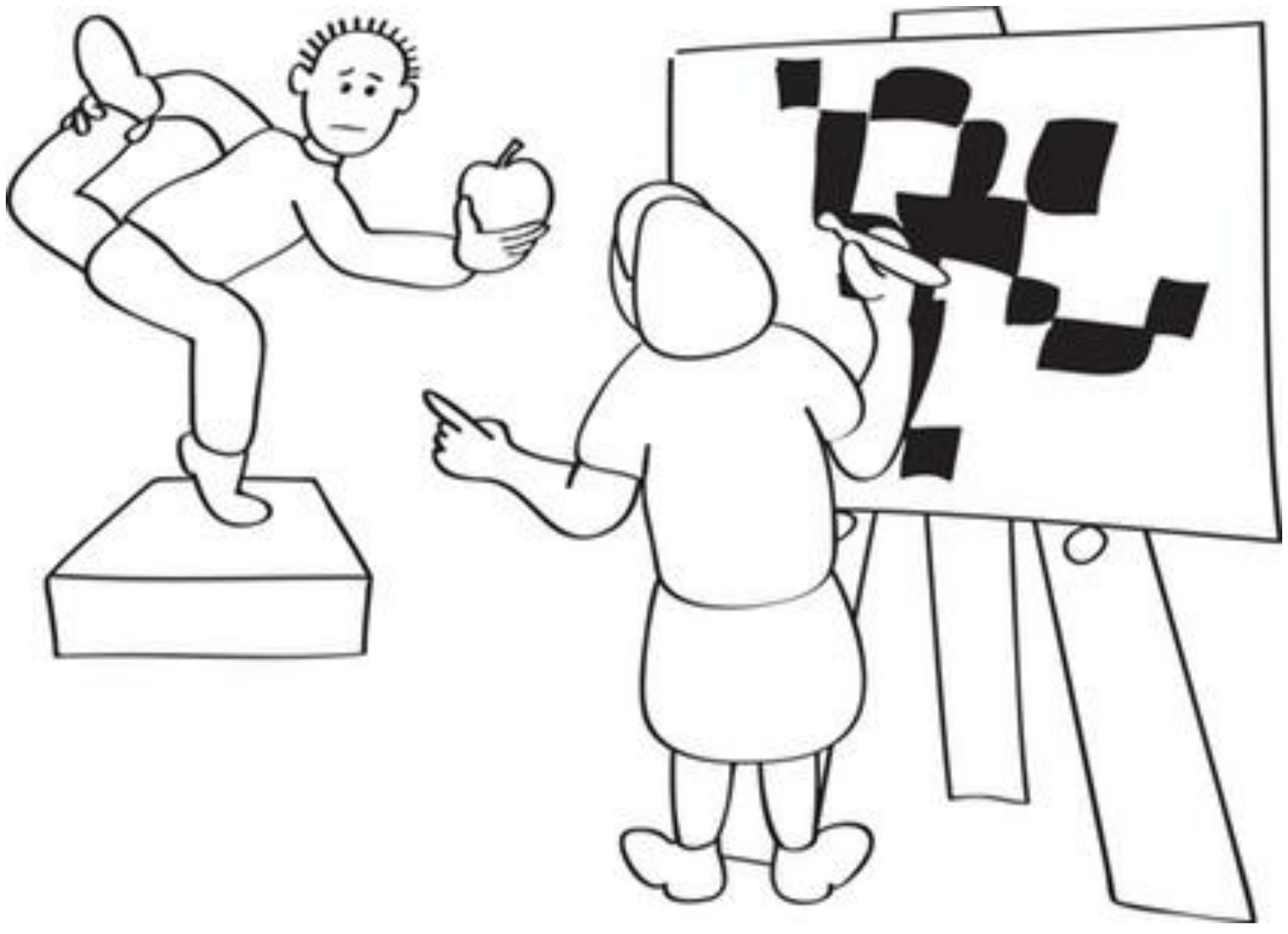


Selective Coding consists in refining the categories and relations, and identify the core category to which all others are related

Selective Coding

- The “core” category is determined and the relationships between it and secondary categories are posited. Core and secondary category relationships are validated later

- Phase that aims to reach a theoretical saturation



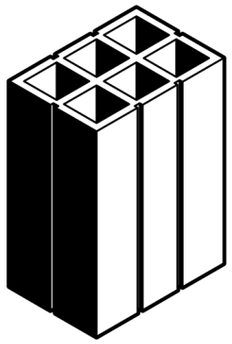
Theory Representation

Representing the Theory

- Theories should be useful instead of being purely results of an academic exercise
- Sjøberg's framework[7] to represent and describe the theory
- Categorization
- Evaluation

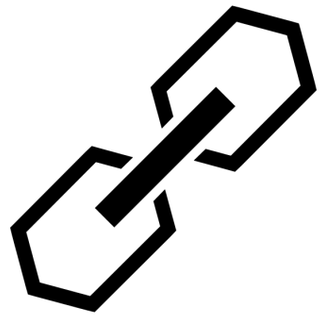
7. J. E. Hannay, D. I. K. Sjoberg, and T. Dyba. 2007. A Systematic Review of Theory Use in Software Engineering Experiments

Sjøberg's framework



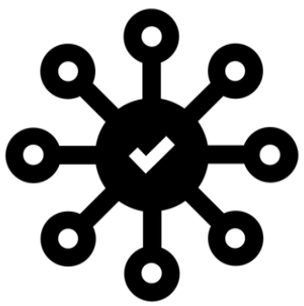
Construct is a basic particle that composes a theory

- categories identified in the axial and selective coding



Proposition is an interaction among constructs

- it comprises the relations established among the categories



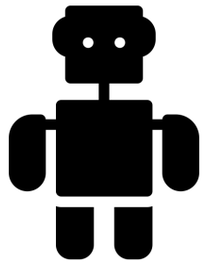
Explanation comprises the factors behind the propositions

- all data

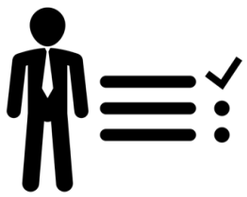
Scope is the universe to which the theory is applicable

Sjøberg's framework

Actor



Technology



Activity

Software System

- The typical SE situation is that an *actor* applies *technologies* to perform certain *activities* on an (existing or planned) *software system*

Identifying Design Problems in the Source Code

A Grounded Theory

Leonardo Sousa
Anderson Oliveira
Willian Oizumi
Simone Barbosa

Alessandro Garcia
Jaejoon Lee
Marcos Kalinowski
Rafael de Mello

Baldoino Neto
Roberto Oliveira
Carlos Lucena
Rodrigo Paes

PUC RIO

LES | DI | PUC-Rio - Brasil

OPUS Research Group

25%

of discussions in a project are about design[1]

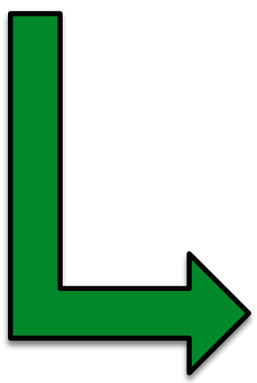
Software design is a fundamental concern during the software development process

1. Brunet et al. 2014. *Do Developers Discuss Design?* (MSR)

Design Decisions

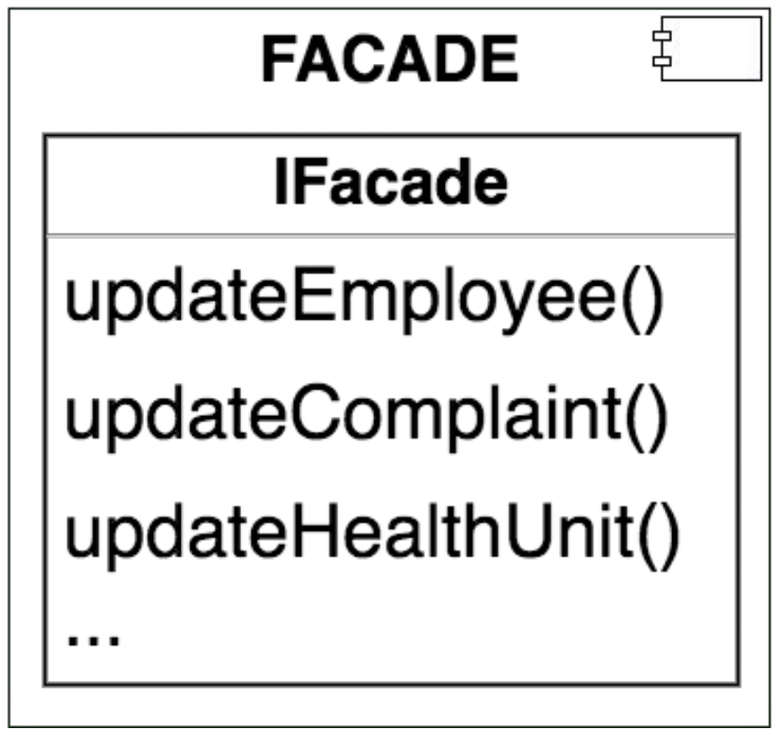
Decisions that affect the system **positively**

Decisions may have a **negative impact** on non-functional requirements

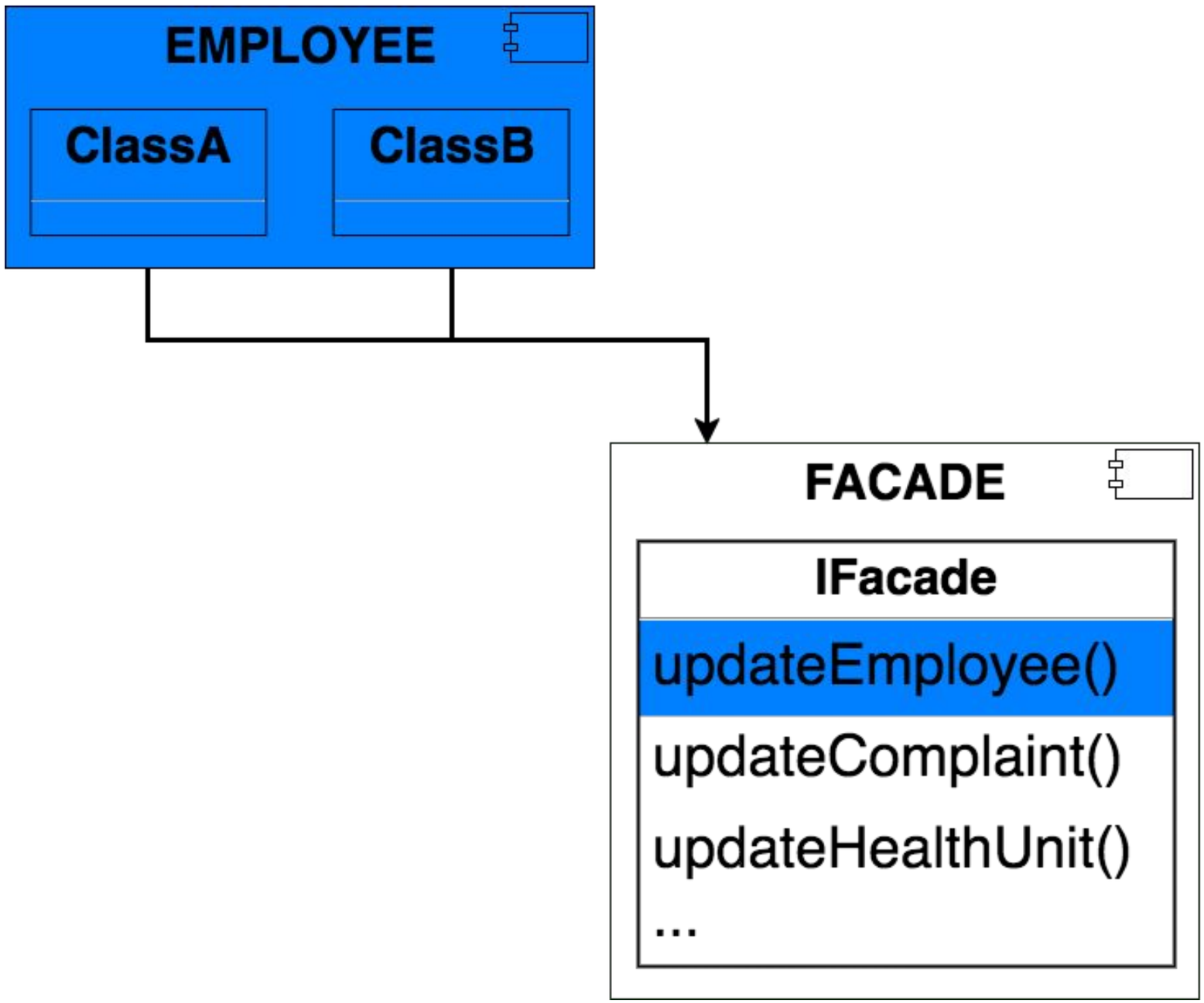


Design Problem is the result of inappropriate design decisions that negatively impact non-functional requirements

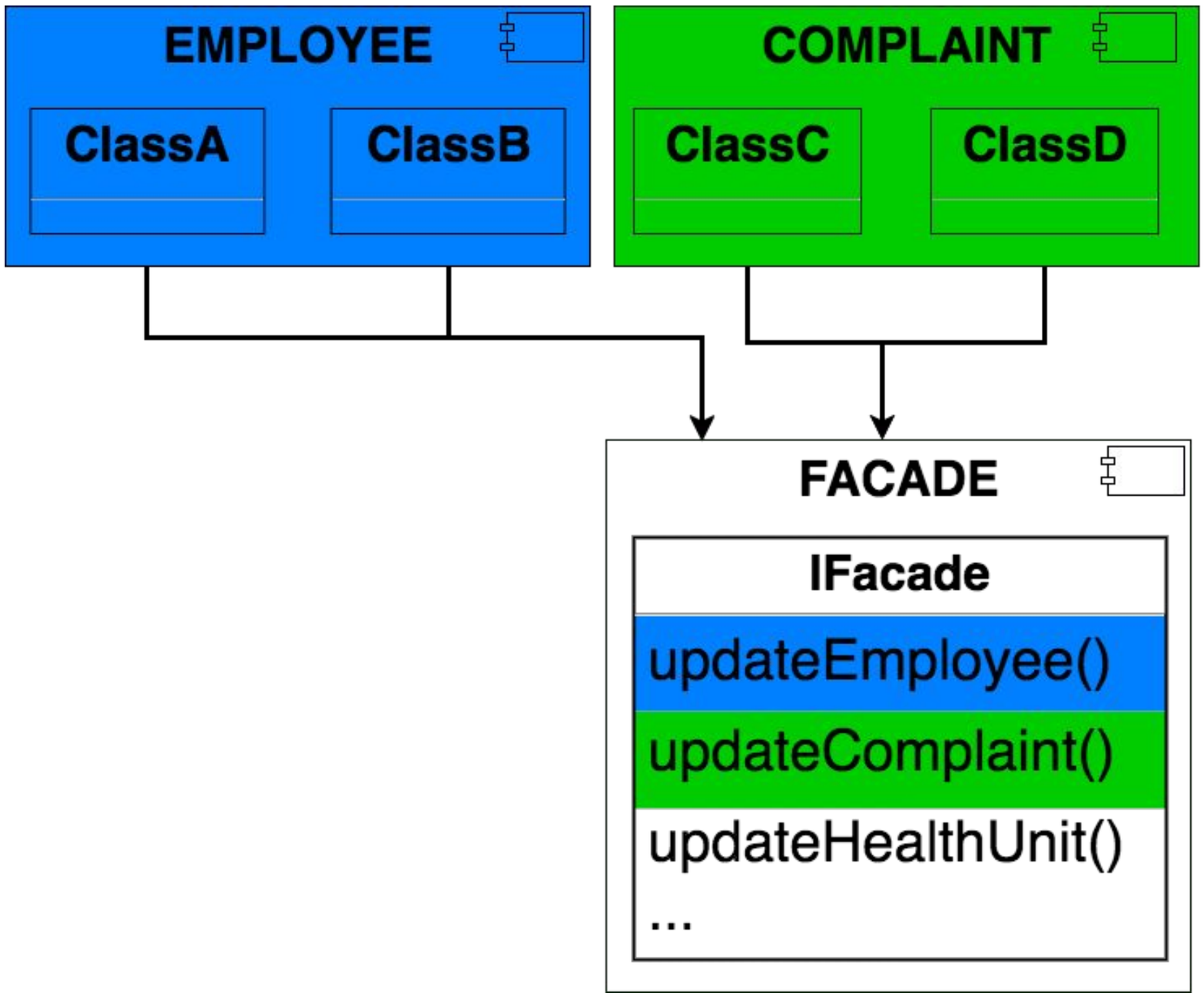
Example of Design Problem



Example of Design Problem

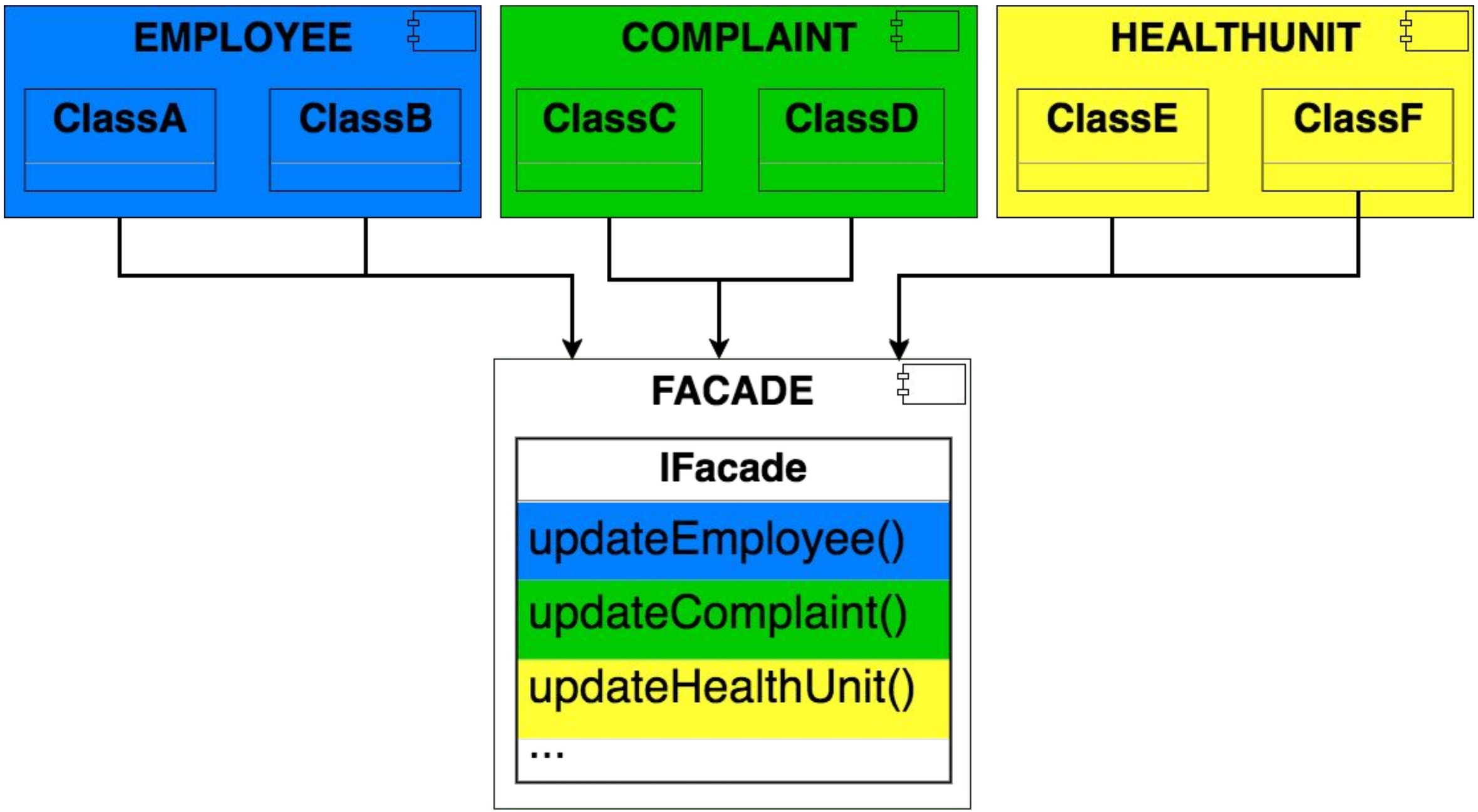


Example of Design Problem



Example of Design Problem

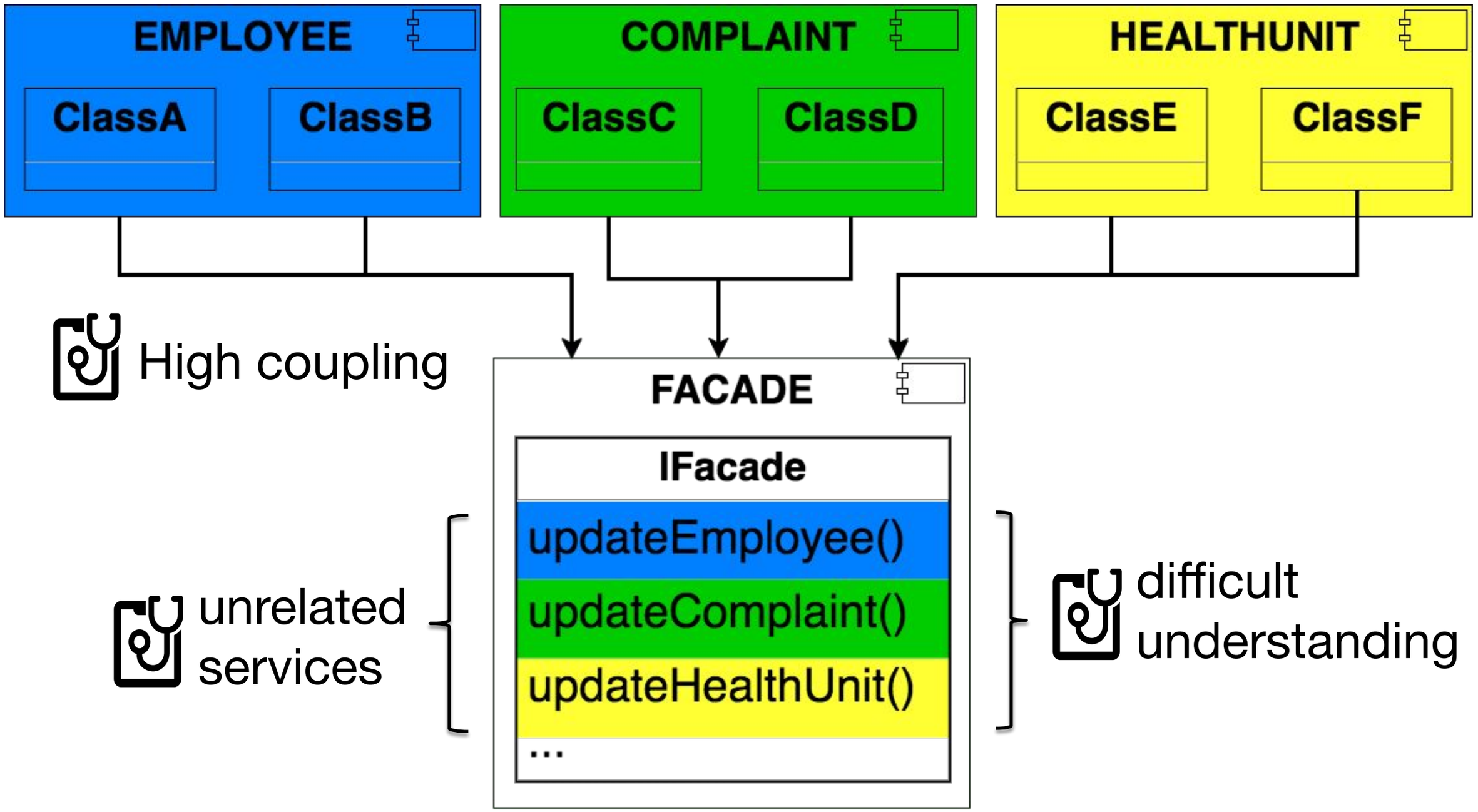
Fat Interface



Symptom is a partial sign or indication of the presence of a design problem

Design Problem Symptoms

encapsulation violation

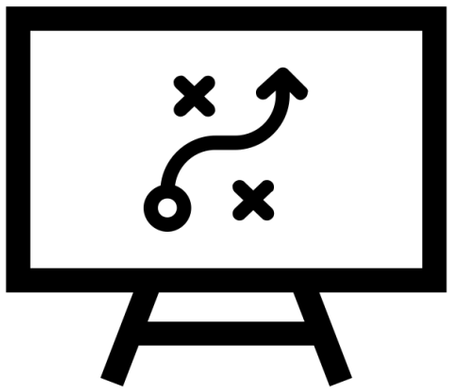


High coupling

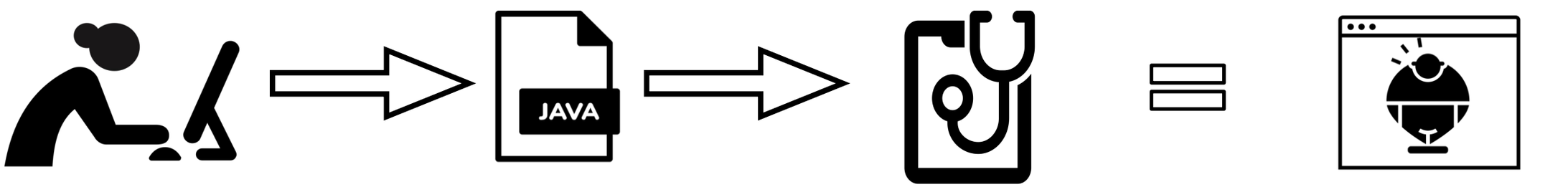
unrelated services

difficult understanding

Investigating the Design Problem Identification



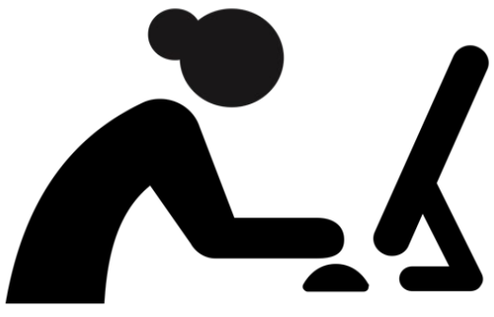
RQ: how do developers identify design problems in source code?



Multi-trial Industrial Experiment

Characterization

Training



Identification

Follow-up

Code Smells

Design Patterns

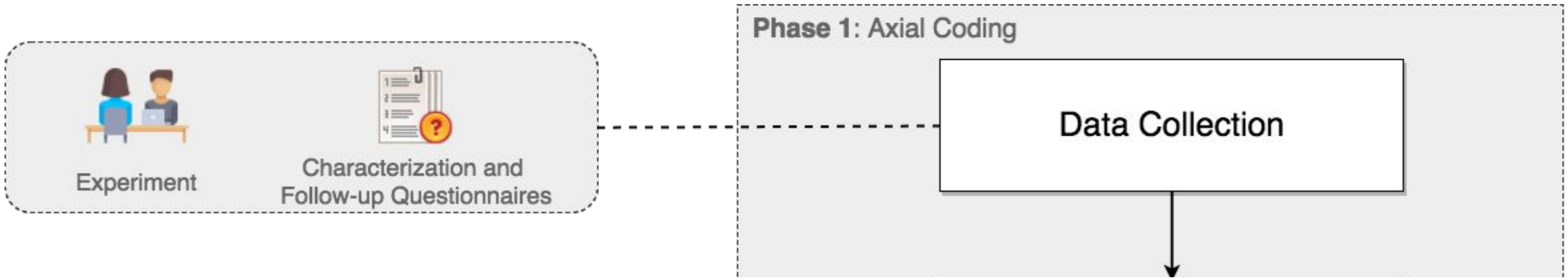
Design Principles

Quality Attributes


Experiment
Characterization and Follow-up Questionnaires
Phase 1: Axial Coding
Data Collection
Transcription
Open Coding
Axial Coding
Raw Transcription
D11: "The symptoms suggest a possible design problem. However, none of them should be rigid rules. Often, as it has been observed in this experiment, it makes sense to have long methods, message chains or many parameters (in the method). In some cases, we could replace a long string of conditional (statements), but it would make it difficult to understand. A method was considered long, but its readability was very clear, which did not justify a refactoring."
Examples of Open Coding
Code 1:
Developer mentions that the class readability is awful
Code 2:
Developer mentions that there is no way to escape from the analyzed implementation
Code 3:
Developer mentions that the analyzed implementation is the standard implementation
Code 4:
Developer accepts that the class is hard to read
Examples of Axial Coding
Category 1:
Analysis of non-functional requirement
Category 2:
Explanation for the existence of the symptoms
Experiment
Characterization and Follow-up Questionnaires
Phase 2: Refining the Axial Coding
Data Collection
Selective Coding
Determining theoretical saturation
Phase 3: Developing the theory
Writing up the theory

Data Collection

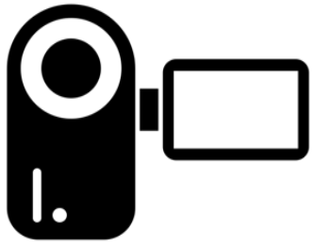
Phase 1



Collecting Data

Phase 1

Think-aloud Method

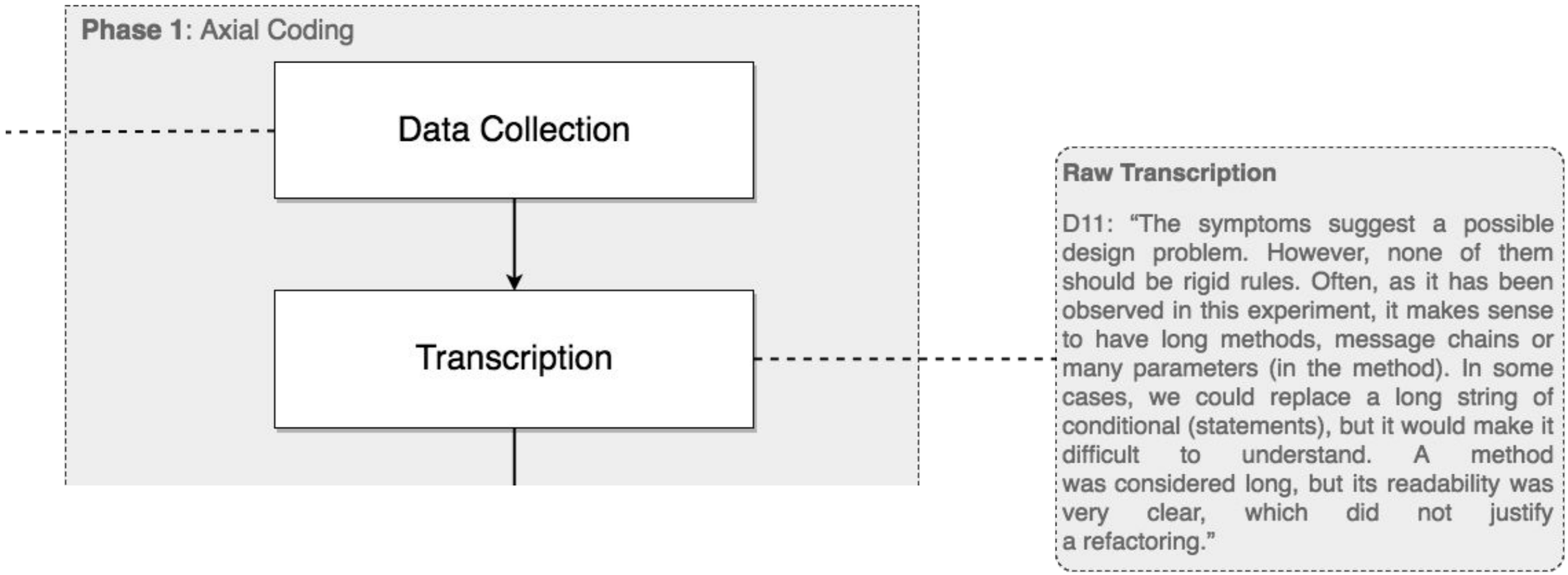


Audio and Video records

Grounded Theory procedures

Data Transcription

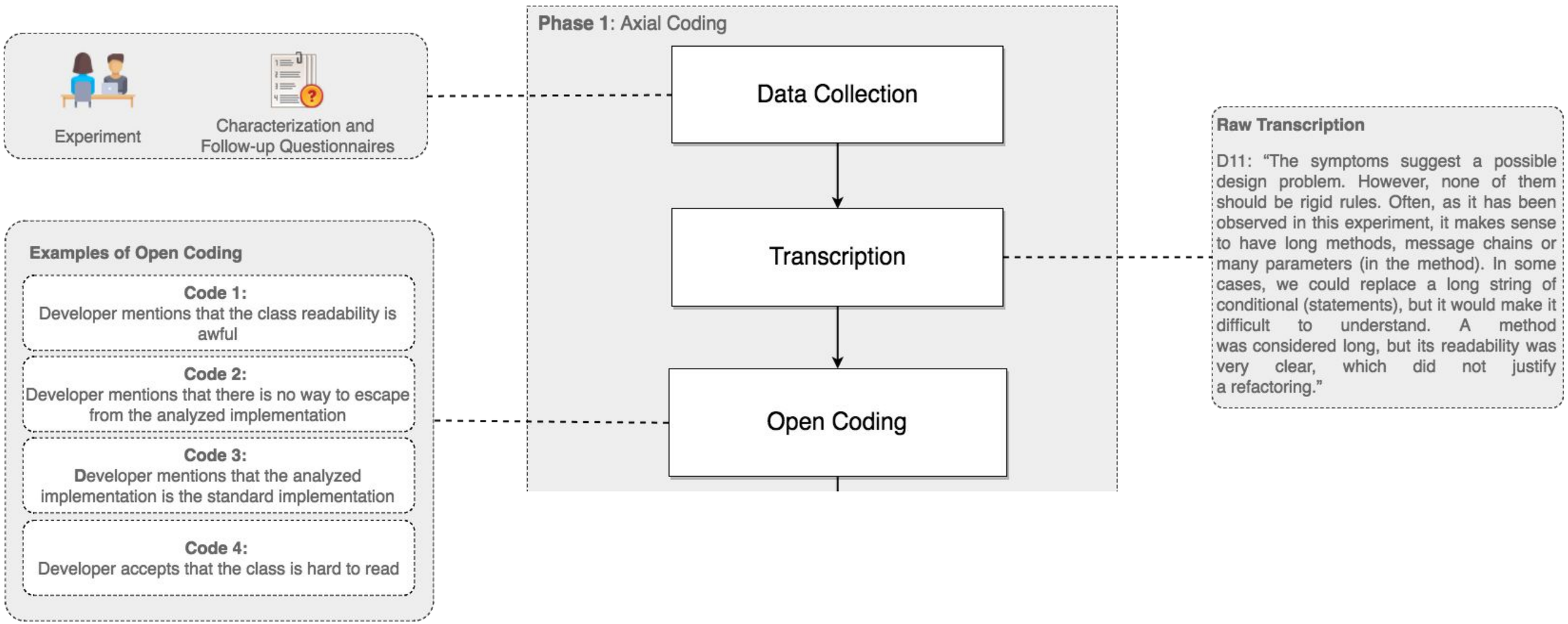
Phase 1



- **Raw Transcript**: "*D6: The readability here is awful, but there is no way to escape from this (implementation). That is the standard (implementation). (...) indeed, it (the class) is not easy to ready*"

Open Coding

Phase 1



- **Raw Transcript**: "*D6: The readability here is awful, but there is no way to escape from this (implementation). That is the standard (implementation). (...) indeed, it (the class) is not easy to ready*"

Open Coding

Phase 1

Raw Transcript: "*D6: The readability here is awful, but there is no way to escape from this (implementation). That is the standard (implementation). (...) indeed, it (the class) is not easy to ready*"

- **Code 1:** developer mentions that the class readability is awful

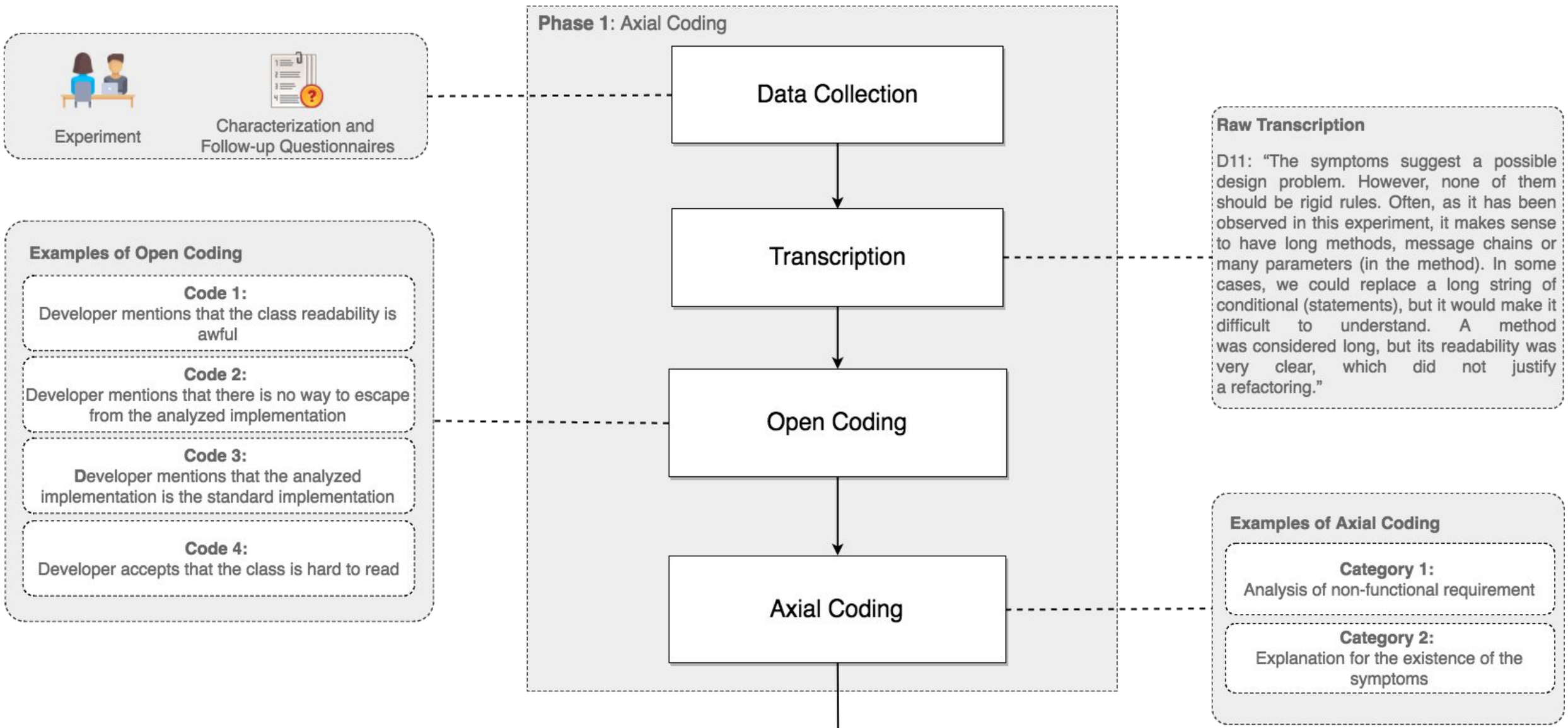
- **Code 2:** developer mentions that there is no way to escape from the analyzed implementation

- **Code 3:** developer mentions that the analyzed implementation is the standard implementation

- **Code 4:** developer accepts that the class is hard to read

Axial Coding

Phase 1



- **Category 1:** analysis of a non-functional requirement
- **Category 2:** explanation for the existence of the symptom

Axial Coding

Phase 1

Code 1: developer mentions that the class readability is awful

Code 4: developer accepts that the class is hard to read

Category 1: analysis of a non-functional requirement

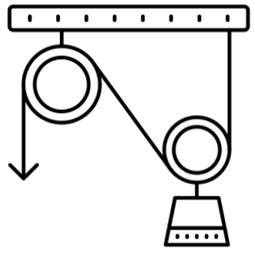
Code 2: developer mentions that there is no way to escape from the analyzed implementation

Code 3: developer mentions that the analyzed implementation is the standard implementation

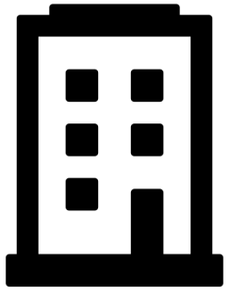
Category 2: explanation for the existence of the symptom

Data Collection

We did not achieve the Theoretical Saturation



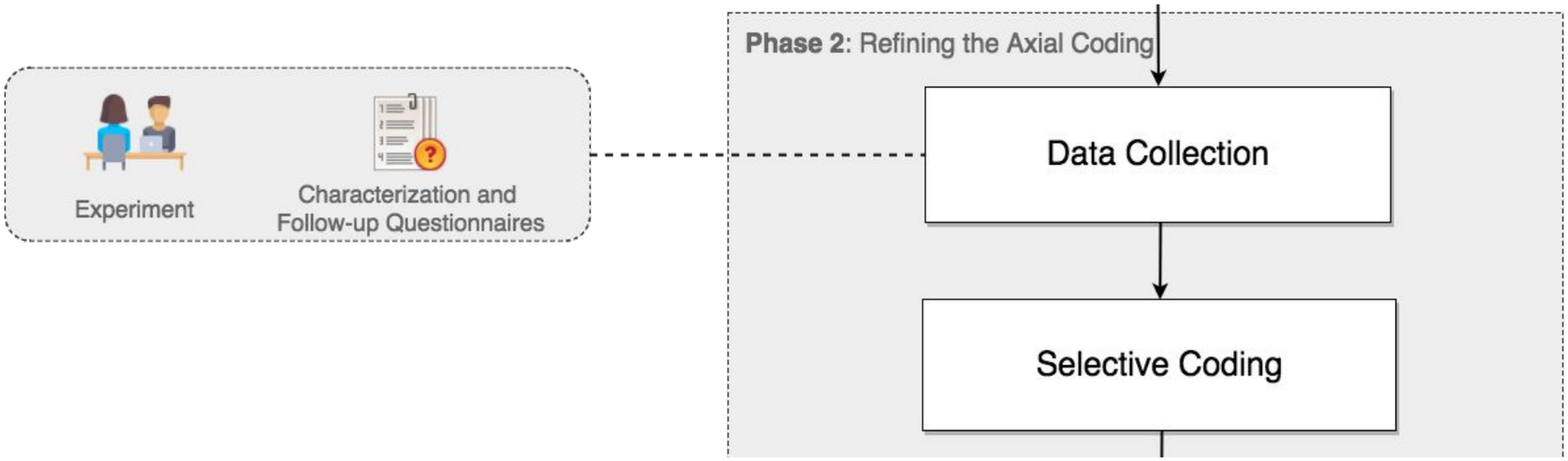
We had to conduct more experiments



We ran the experiments with two more companies

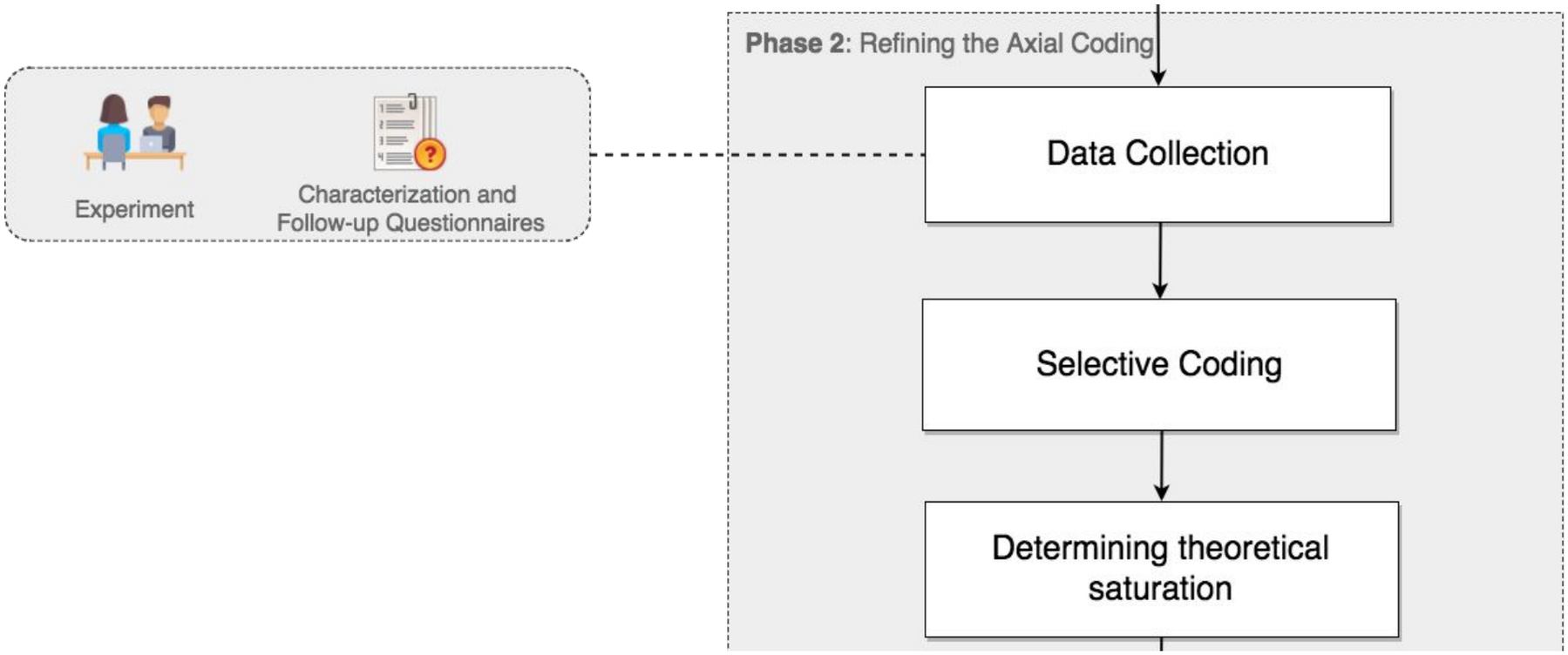
Selective Coding

Phase 2



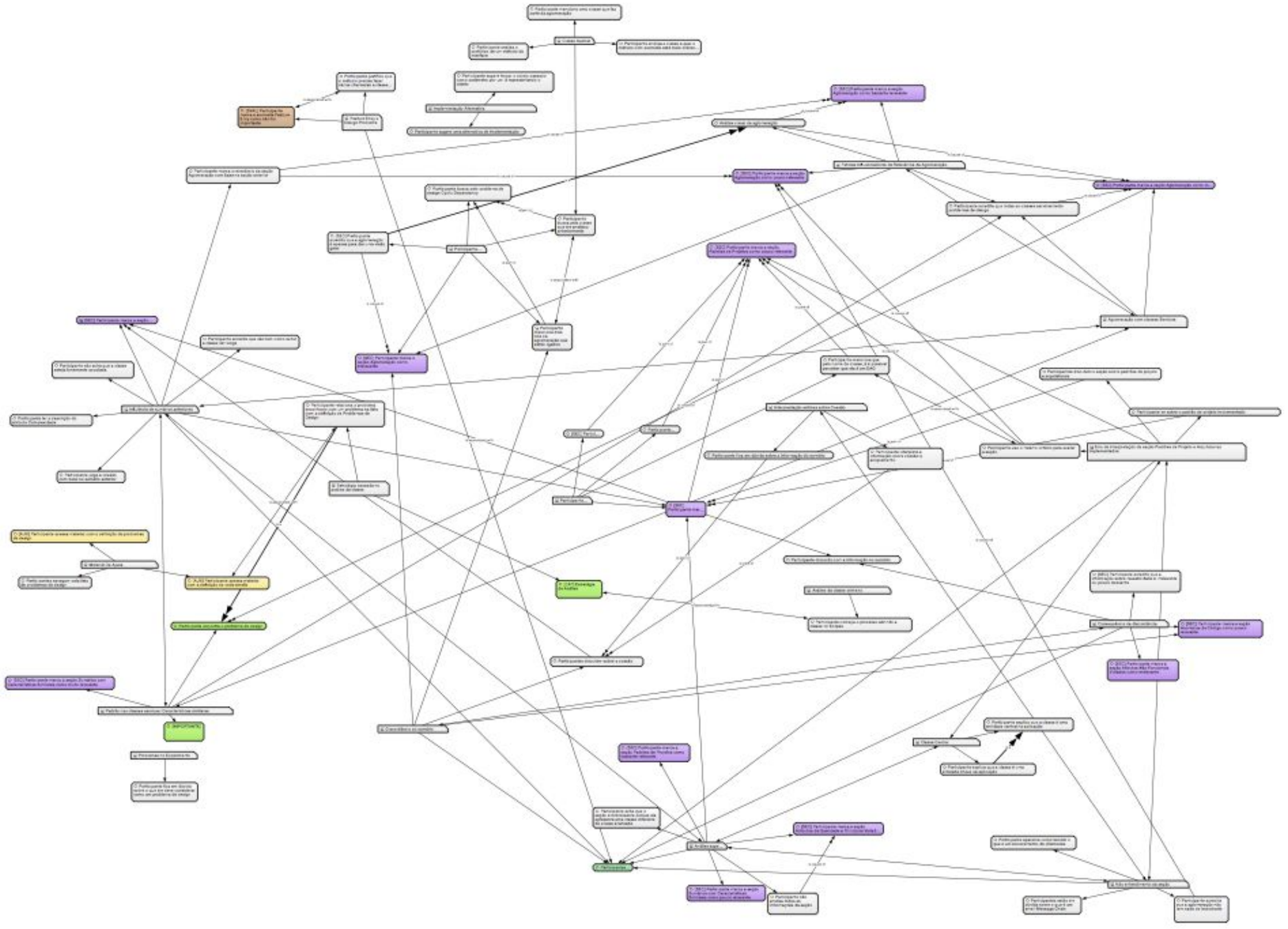
Determining Theoretical Saturation

Phase 2



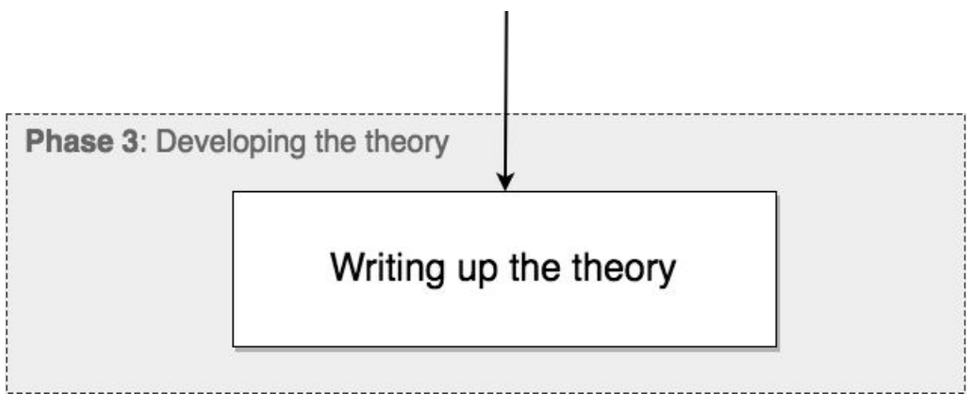
Networks: Graphic Notation

Phase 2



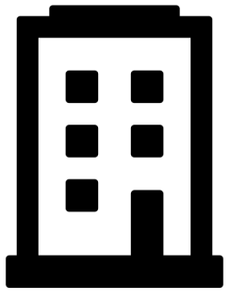
Writing up the Theory

Phase 3

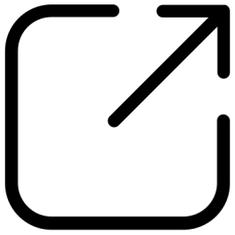


- We had to map the (grounded) theory according to Sjøberg's framework
-
  - Constructs
  - Propositions
  - Explanations
  - Scope

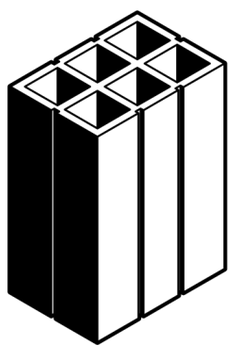
Some Numbers



5 companies, 8 systems and 23 developers



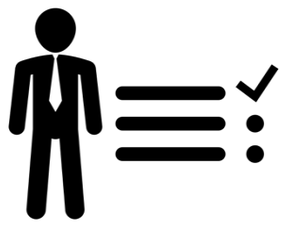
1,161 codes, 9 networks and 16 hours of video



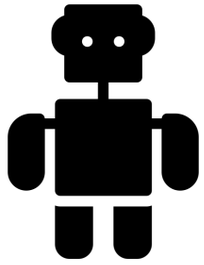
15 constructs and 18 propositions

Theory according to Sjøberg's framework

Actor: **Software Developer**



Activity: **Identification of Design Problems**

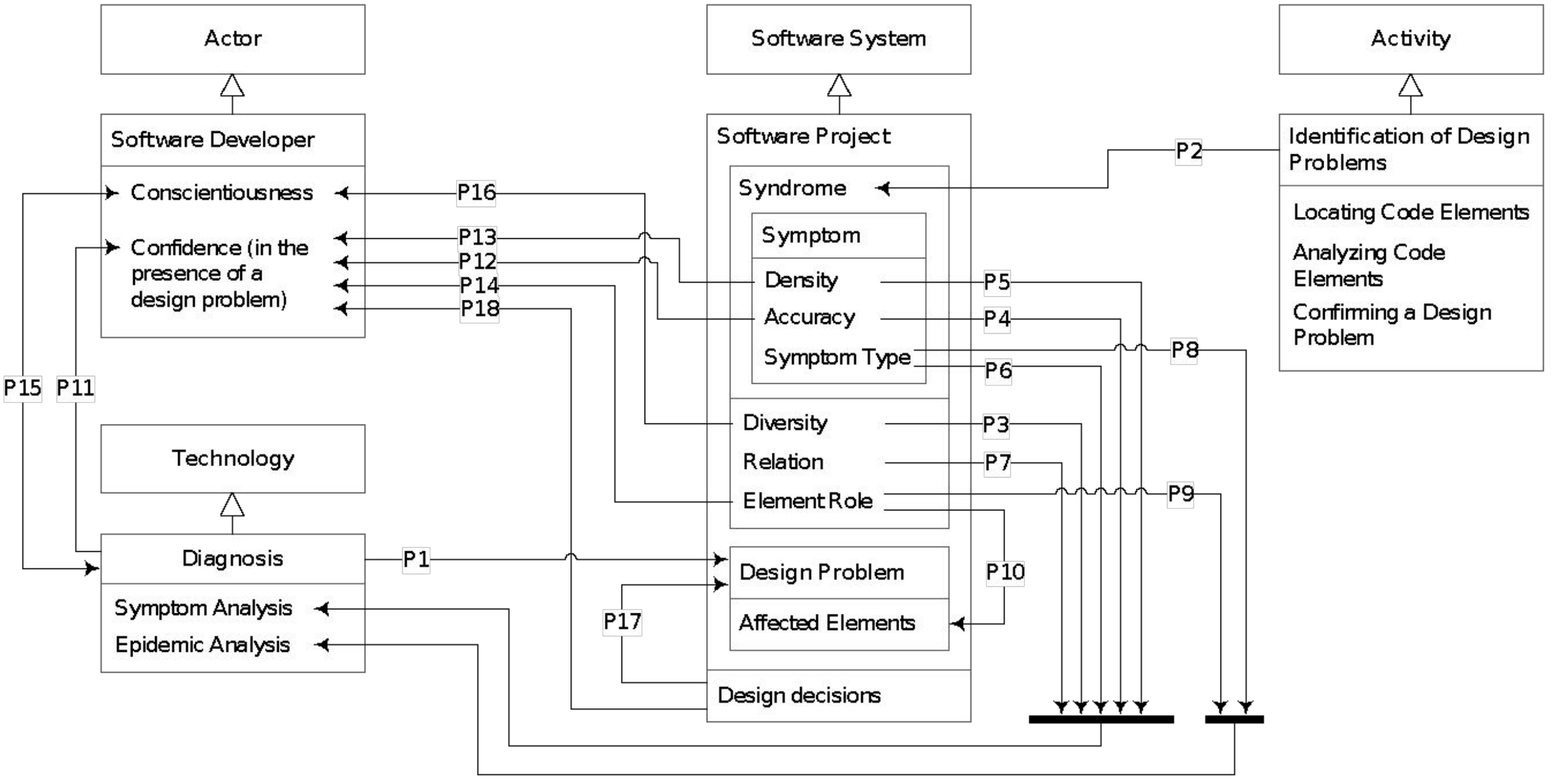


Technology: **Diagnosis**

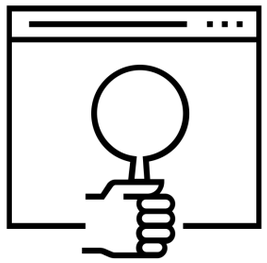
Software System: **Source Code**

The theory is supposed to be applicable in systems in which developers intend to identify design problems by analyzing symptoms that manifest themselves in the source code

A Theory of Design Problem Identification



Steps to Identify Design Problems



Locating elements

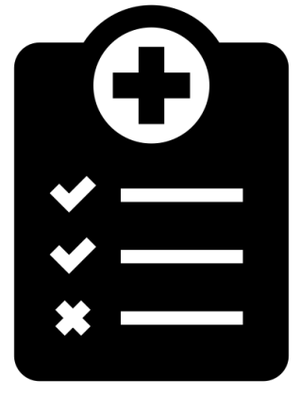
Analyzing elements

Confirming the problem

Focusing on Specific Steps

Developers rely on Multiple Symptoms

- Code Smells
- Violation of Object-Oriented Principles
- Violation of Architectural and Design Patterns
- Poor Structural Quality Attributes
- Violation of Non-functional Requirements



Symptom Helpfulness

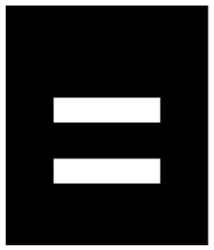
Characteristics that developers consider when they choose the symptoms most likely to help them

Type, Accuracy and Density

Relation and Diversity

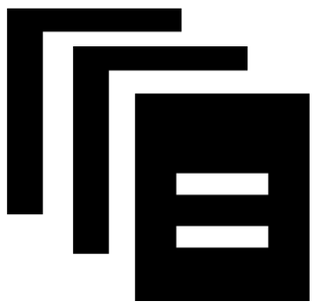
Prioritizing Symptoms with these Characteristics

Design Problem Diagnosis



Symptom Analysis

- Analyzing a set of symptoms affecting the same element
- Combining multiple related symptoms

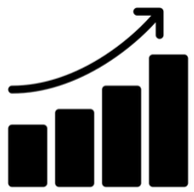


Epidemic Analysis

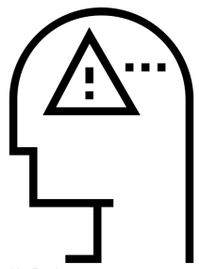
- Analyzing other elements with a similar set of symptoms
- Prioritizing key elements

Automating the diagnosis

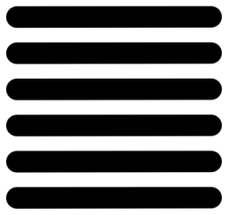
Developers' Factors



Increasing the developers' confidence



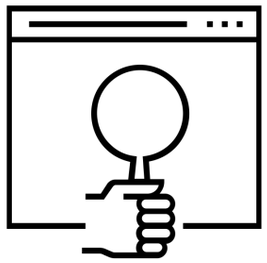
Increasing the developers' conscientiousness



Justifying the presence of a design problem

Reducing "human factors"

Improving Design Problem Diagnosis

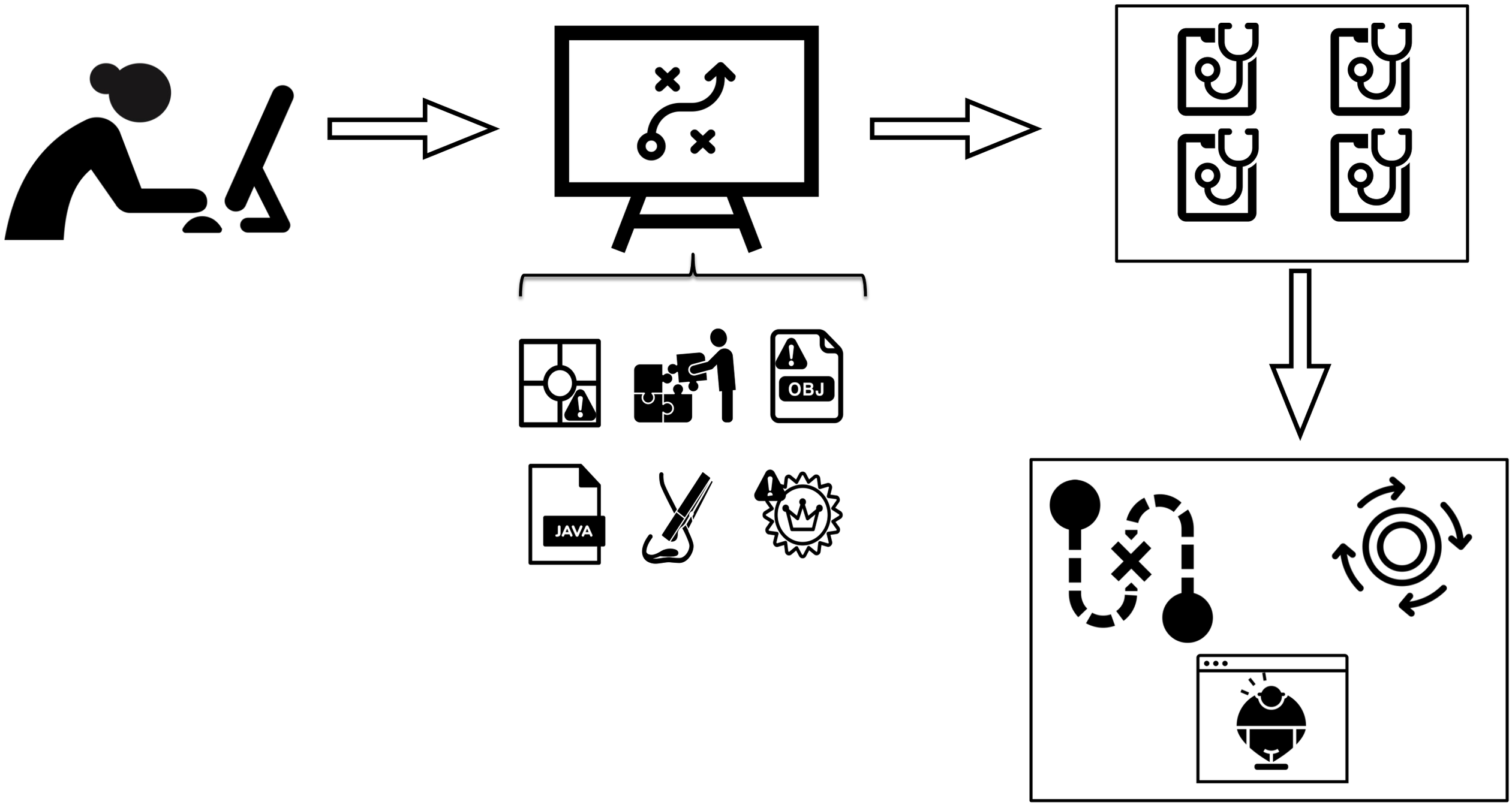


Supporting Multiple Symptoms

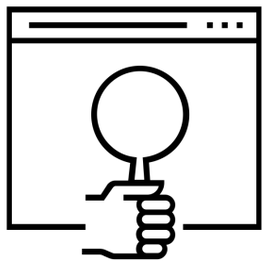
Personalizing the Filter and Detection of Symptoms

Visualization Support

Visualization Support



Concluding Remarks



Mature areas rely on (and refine) theories to further advance the field[9]

Theory describing the activities and factors that influence on how developers identify design problems

Solutions that emerged from the theory and can improve design problem identification

9. Klaas-Jan Stol and Brian Fitzgerald. 2015. *Theory-oriented software engineering.*

Further Reading

- Sousa, Leonardo, et al. "Identifying design problems in the source code: a grounded theory." 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). IEEE, 2018.

- Sousa, Leonardo. Understanding How Developers Identify Design Problems in Practice. 2018. Tese de Doutorado. PUC-Rio.

- Lazar, Jonathan, Jinjuan Heidi Feng, and Harry Hochheiser. Research methods in human-computer interaction. Morgan Kaufmann, 2017.

Identifying Design Problems in the Source Code

A Grounded Theory

Leonardo Sousa
Anderson Oliveira
Willian Oizumi
Simone Barbosa

Alessandro Garcia
Jaejoon Lee
Marcos Kalinowski
Rafael de Mello

Baldoino Neto
Roberto Oliveira
Carlos Lucena
Rodrigo Paes

PUC RIO

LES | DI | PUC-Rio - Brasil

OPUS Research Group

Backup Slides

A Brief Story

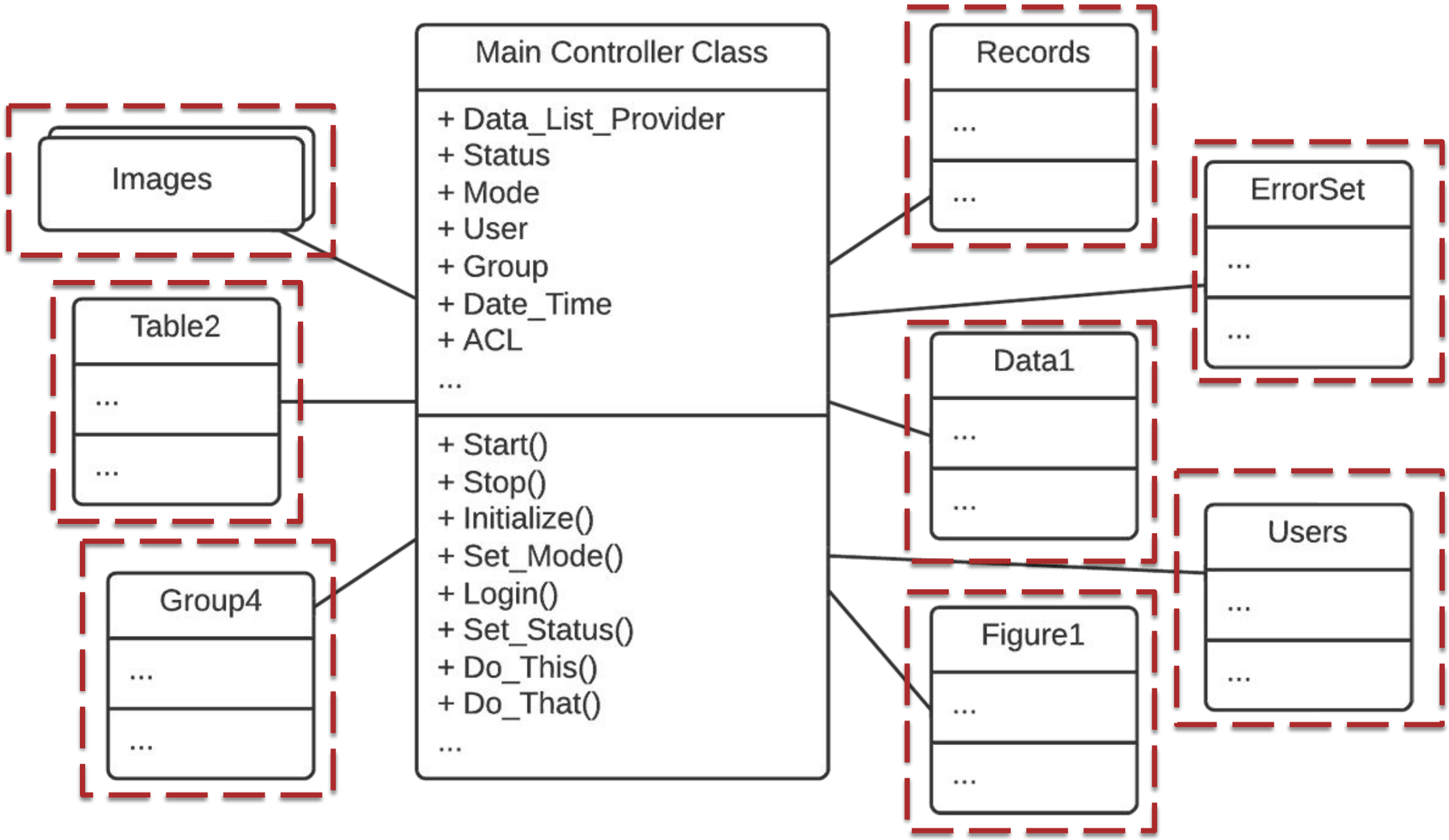
Code Smell is a recurring structure in source code that may indicate a deeper problem in a software system[1]

1. M Fowler. 1999. *Refactoring: Improving the Design of Existing Code.*

God Class

Example of Code Smell

When a class centralizes the system functionality



Collaborative Identification of Code Smells

Individuals

VS

Pairs

Groups

Does collaboration affect the effectiveness of code smell identification?

Quantitative Results

Pairs

VS

Individuals

	Program A	Program B	Program C
Individuals	3.33	1.38	6.88
Pairs	7.25	5	14

Why?

Seeking for Answer

We can analyze the questionnaire

Do you believe that the collaboration helped on the identification of code smells? Justify

- *Yes. Collaboration have as strengths the communication among members and the possibility of a more precise analysis because 'n' eyes see more than two* [...]

Moreover, 100% of participants believe they have identified more smells by working collaboratively

Quantitative Results

Effectiveness?

Groups

VS

Individuals

	Program A	Program B	Program C
Individuals	3.33	1.38	6.88
Groups	6	3	5

Why?

Lack Understanding About the Results

	Program C
Individuals	6.88
Groups	5

Why?

Key Components and Concepts (1/5)

- **Limit exposure to literature**: avoid comprehensive literature review
- **Treat everything as data**: quantitative data, videos, pictures, diagrams…
- **Immediate and continuous data analysis**: data collection and analysis are simultaneous

Key Components and Concepts (2/5)

- **Theoretical sampling**: researcher identifies further data sources based on gaps in the emerging theory or to further explore unsaturated concepts

- **Theoretical sensitivity**: ability to conceptualize, and to establish relationships between concepts

- **Coding**: process to labeling ‘incidents’ and their properties in the data

Key Components and Concepts (3/5)

- **Concepts:** collections of codes of similar content that allows the data to be grouped
- **Categories:** broad groups of similar concepts that are used to generate a theory
- **Memoing**: researcher writes memos (e.g. notes, diagrams, sketches) to elaborate categories, describe preliminary properties and relationships between categories, and identify gaps

Key Components and Concepts (4/5)

- **Constant comparison**: researcher constantly compares data, memos, codes and categories
- **Memo sorting**: continuous process of oscillating between the memos and the emerging theory outline to find a suitable fit for all categories that resulted from the coding

Key Components and Concepts (5/5)

- **Cohesive theory**: researcher attempts to move beyond superficial categories and develop a cohesive theory of the studied phenomenon

- **Theoretical saturation**: the point at which a theory's components are well supported and new data is no longer triggering revisions or reinterpretations of the theory