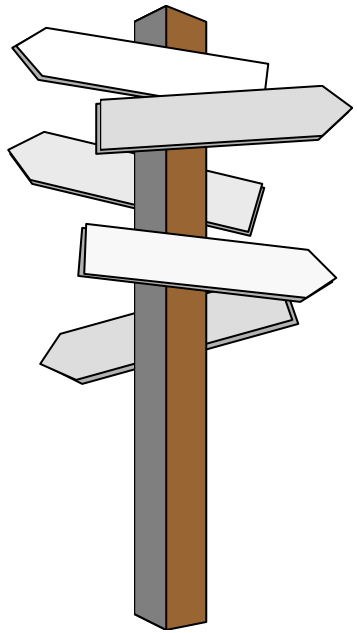
# Design Extraction

# Goals of this Lecture

- Design is not code with boxes and arrows
- Design extraction is not trivial
- Design extraction should scale up
- Design extraction can be supported by computers but not fully automatize
- Give a critic view on hype: “we read your code and produce design”
- Fertilize you with some basic techniques that may help you
- Show that UML is not that simple and clear but still really useful

# Outline

- Why Extracting Design? Why Uml?
- Basic UML Static Elements
- Interpreting UML
- Language Specific Issues
- Tracks For Extraction
- Extraction Techniques
- Conclusion

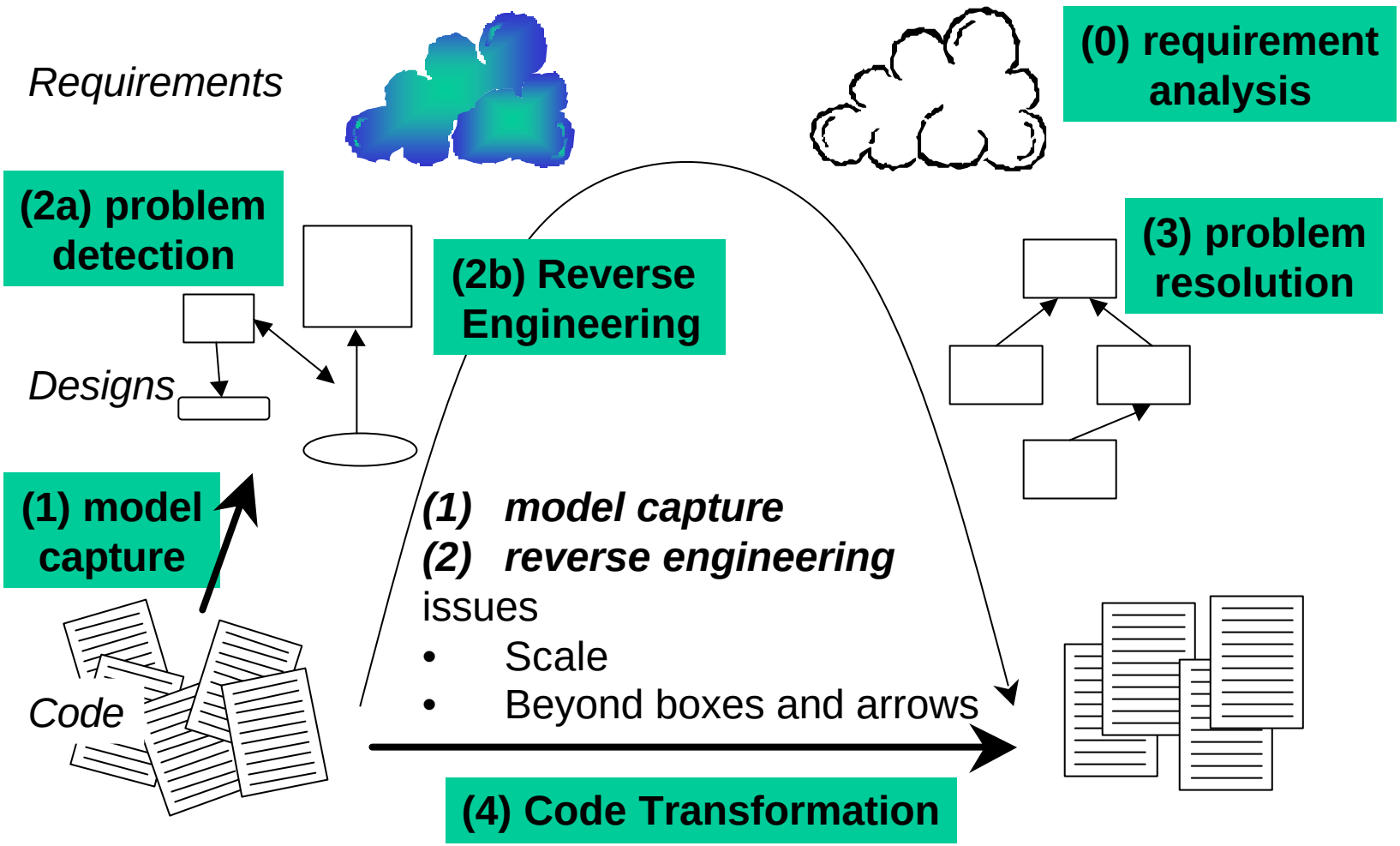


# Why Design Extraction is Needed?

- Documentation inexistent, obsolete, or too prolix
- Abstraction needed to understand applications
- Original programmers left
- Only the code available

- Why UML?
  - Standard
  - Communication based on a common language
  - Can support documentation if we are precise about its interpretation
  - Extensible
    - stereotypes

# The Reengineering Life-Cycle



# War story: “Company X is in trouble.”

Their product is successful (they have 60% of the world market).

But:

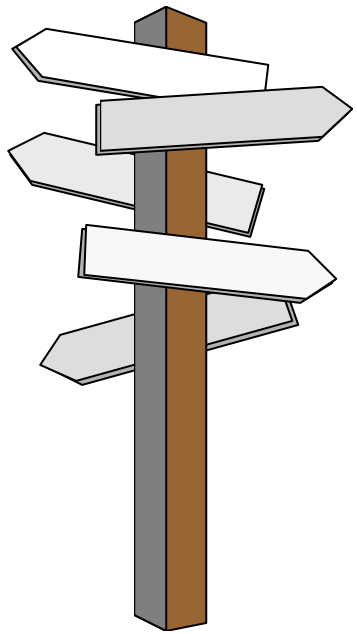
- all the original developers left,
- there is no documentation at all,
- there is no comment in the code,
- the few comments are obsolete,
- there is no architectural description,...

... and they must change the product to take into account new client requirem.

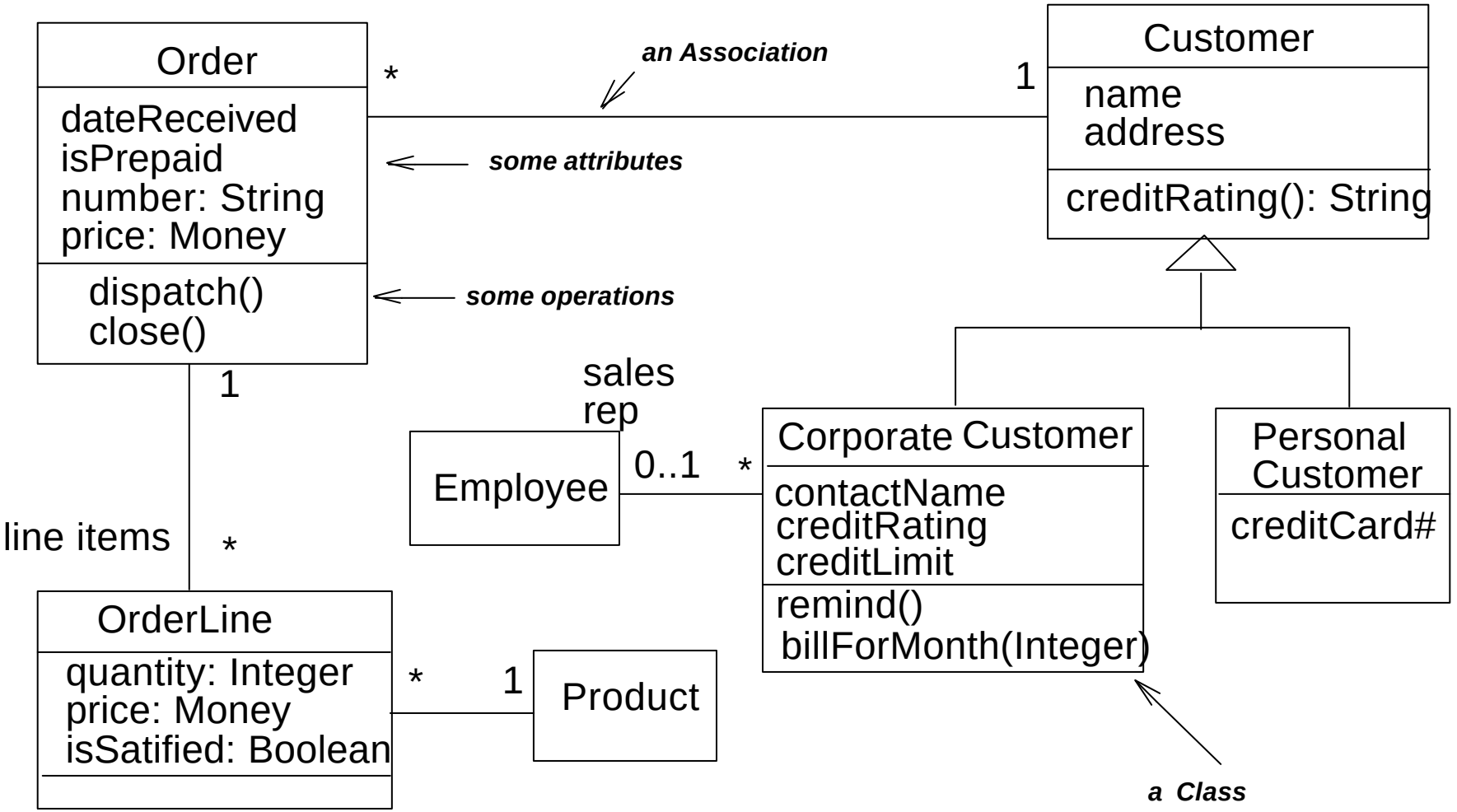
- They asked a student to reconstruct the design. ;-)

# RoadMap

- Why Extracting Design? Why Uml?
- Basic UML Static Elements
  - Experimenting With Extraction
- Interpreting Uml
- Language Specific Issues
- Tracks For Extraction
- Extraction of Intention
- Extraction For The Reuser
- Extraction of Interaction
- Conclusion



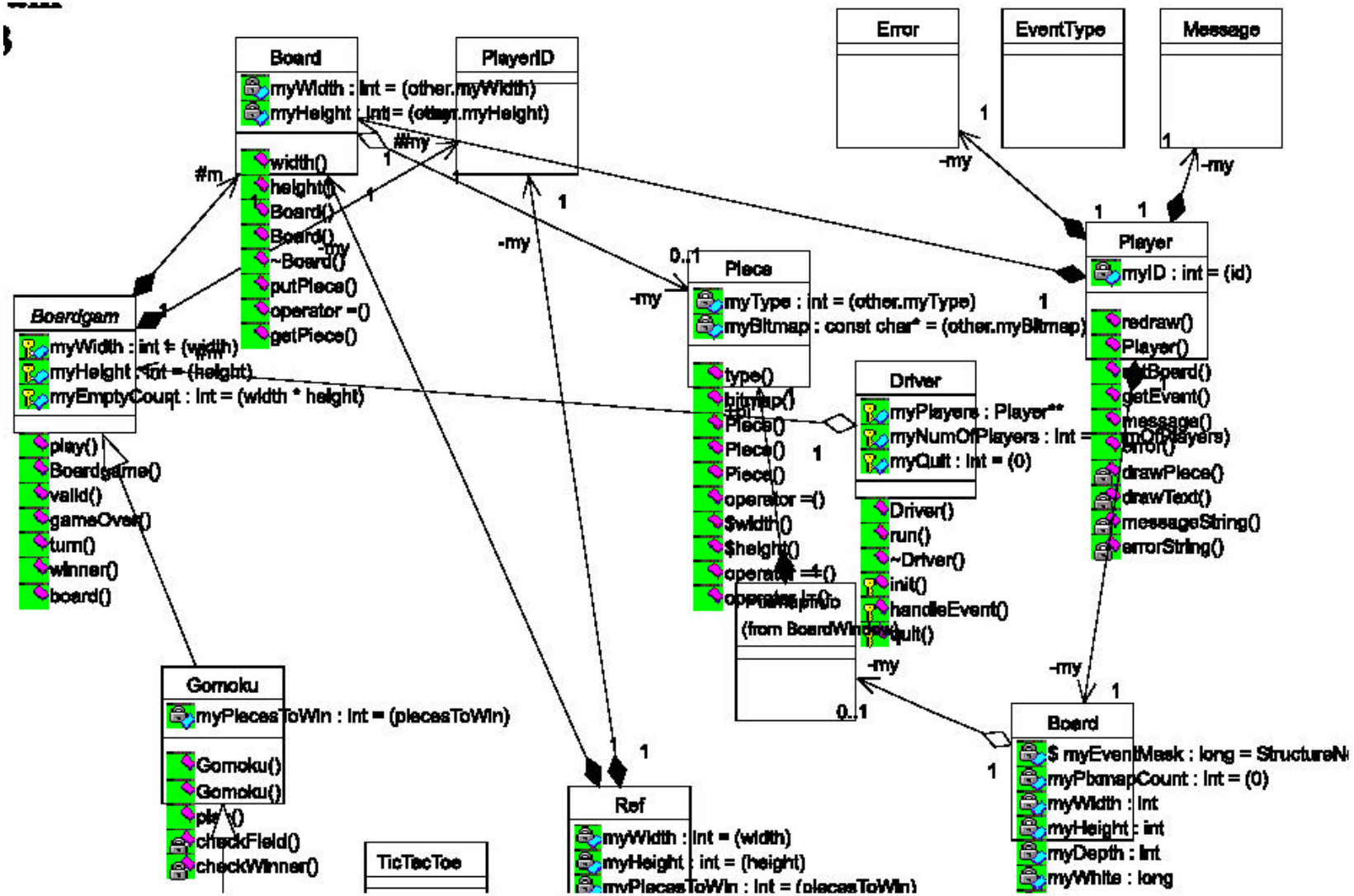
# The Little Static UML



# Let us practice!

- A small example in C++: A Tic-Tac-Toe Game!
- You will do it now.....
- But:
  - do not interpret the code
  - do not make any assumption about it
  - do not filter out anything

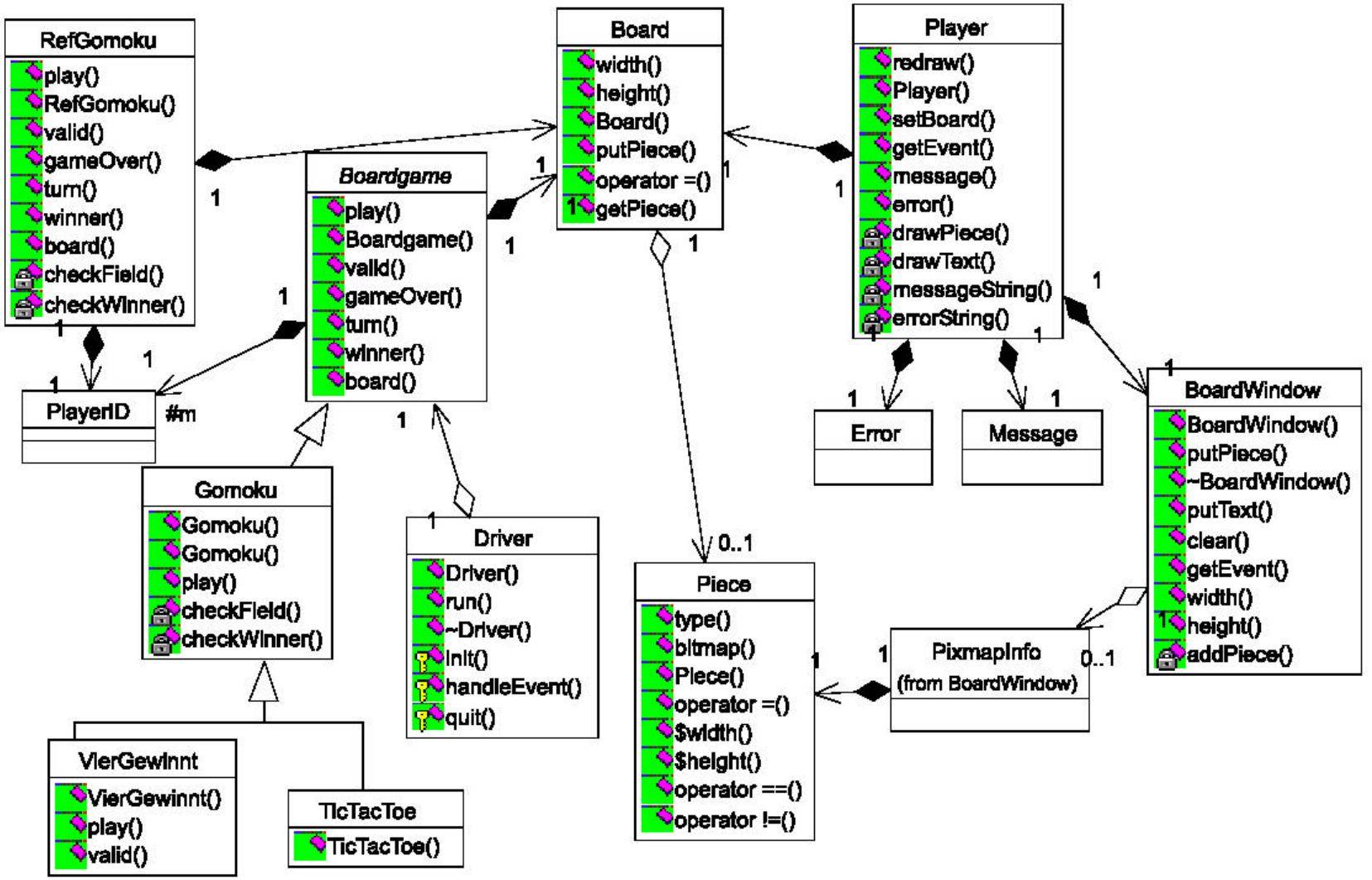
# A First View


Board
myWidth : Int = (other.myWidth)
myHeight : Int = (other.myHeight)
width()
height()
Board()
Board()
~Board()
putPiece()
operator =()
getPiece()
PlayerID
Error
EventType
Message
Boardgam
myWidth : int = (width)
myHeight : Int = (height)
myEmptyCount : Int = (width * height)
play()
Boardgame()
valid()
gameOver()
turn()
winner()
board()
Piece
myType : Int = (other.myType)
myBitmap : const char* = (other.myBitmap)
type()
bitmap()
Piece()
Piece()
Piece()
operator =()
$width()
$height()
Driver
myPlayers : Player**
myNumOfPlayers : Int =
myQuit : Int = (0)
Driver()
run()
~Driver()
init()
handleEvent()
quit()
Player
myID : int = (id)
redraw()
Player()
getEvent()
message()
drawPiece()
drawText()
messageString()
errorString()
Gomoku
myPiecesToWin : Int = (piecesToWin)
Gomoku()
Gomoku()
play()
checkField()
checkWinner()
TicTacToe
Ref
myWidth : Int = (width)
myHeight : Int = (height)
Board
myPixmapCount : Int = (0)
myWidth : Int
myHeight : Int
myDepth : Int
myWhite : long
#m
#my
-my
1
0..1

# Evaluation

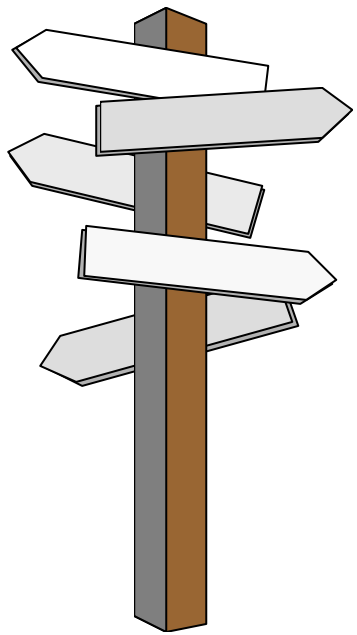
- We should have heuristics to extract the design.
- Try to clean the previous solution you found
- Try some heuristics like **removing**:
  - private information,
  - remove association with non domain entities,
  - simple constructors,
  - destructors, operators

# A Cleaner View



# Roadmap

- Why Extracting Design? Why Uml?
- Basic Uml Static Elements
- **Experimenting With Extraction**
  - **Interpreting Uml**
- Tracks For Extraction
- Extraction Techniques
- Conclusion



# Three Essential Questions

When we extract design we should be precise about:

1. What are we talking about? Design or implementation?

2. What are the conventions of interpretation that we are applying?

3. What is our goal?
   - documentation programmers,
   - framework users,
   - high-level views,

# Levels of Interpretations: Perspectives

- Fowler proposed 3 levels of interpretations called perspectives [Fowl97a]:
  - conceptual
  - specification
  - implementation

- Three Perspectives:
  - Conceptual
    - we draw a diagram that represents the concepts that are somehow related to the classes but there is often no direct mapping.
    - "Essential perspective"
  - Specification
    - we are looking at **interfaces of software** not implementation
    - types rather than classes. Types represent interfaces that may have many implementations
  - Implementation
    - implementation classes

# Attributes in Perspectives

- Syntax:
  - visibility attributeName: attributeType = defaultValue
  - Example: **`+ name: String`**
- Conceptual:
  - Customer name = Customer has a name
- Specification:
  - Customer class is responsible to propose some way to query and set the name
- Implementation:
  - Customer has an attribute that represents its name
- Possible Refinements: Attribute Qualification
  - Immutable: Value never change
  - Read-only: Client cannot change it

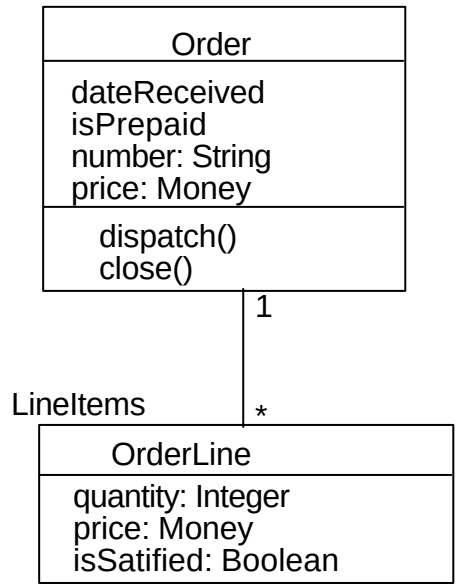
# Operations in Perspectives

- Syntax:
  - visibility name (parameter-list) : return-type
  - Ex: + public, # protected, - private
- Conceptual:
  - principal functionality of the object. It is often described as a sentence
- Specification:
  - public methods on a type
- Implementation:
  - methods
- Possible Refinements: Method qualification:
  - Query (does not change the state of an object)
  - Cache (does cache the result of a computation),
  - Derived Value (depends on the value of other values),
  - Getter, Setter

# Associations

Represent relationships between instances

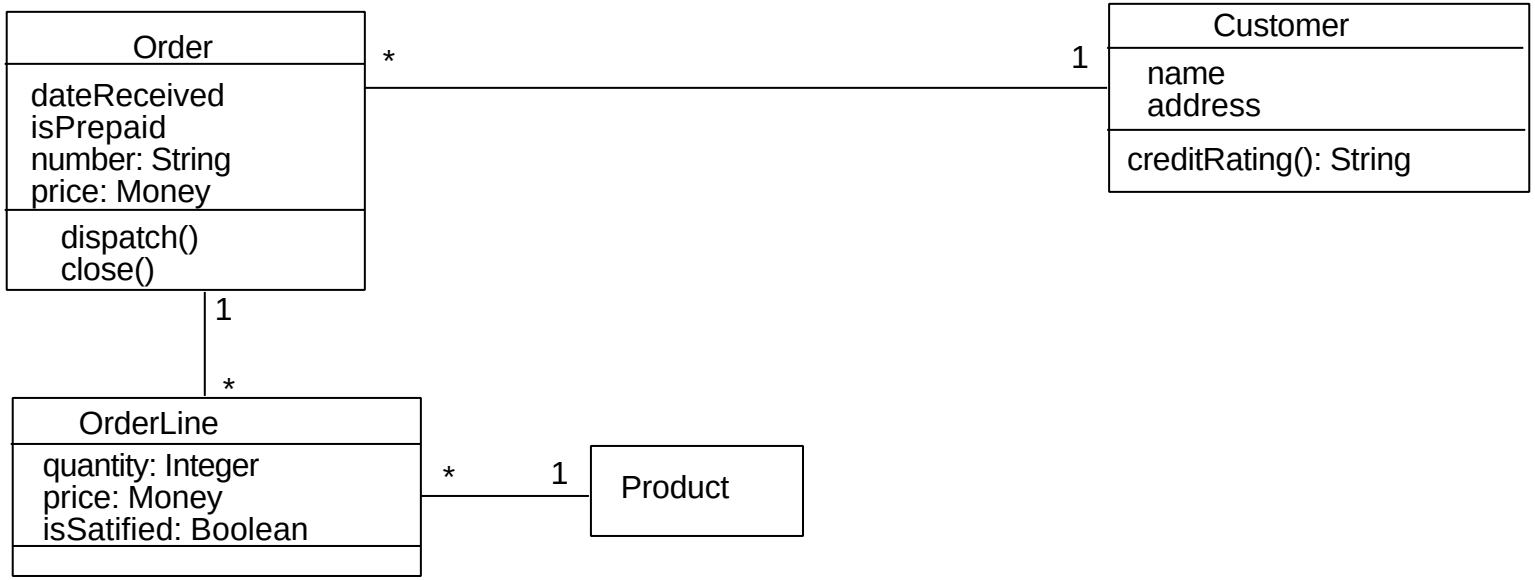
- Each association has two roles: each role is a direction on the association.
- a role can be explicitly named, labelled near the target class
- if not named from the target class and goes from a source class to a target class
- a role has a multiplicity: 1, 0, 1..*, 4
- LineItems =
  - role of direction Order to OrderLines
- LineItems role = OrderLine role
- One Order has several OrderLines



# Associations: Conceptual Perspective

Associations represent conceptual relationships between classes

- An Order has to come from a single Customer.
- A Customer may make several Orders.
- Each Order has several OrderLines that refers to a single Product.
- A single Product may be referred to by several OrderLines.



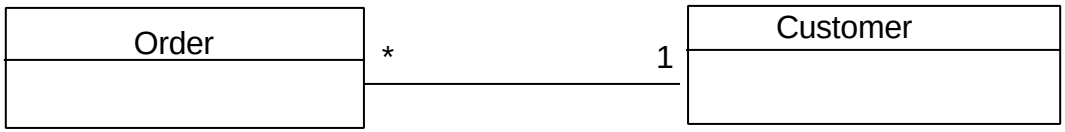
# Associations: Specification Perspective

Associations represent responsibilities

- One or more methods of Customer should tell what Orders a given Customer has made.
- Methods within Order will let me know which Customer placed a given Order and what Line Items compose an Order

Associations also implies responsibilities for updating the relationship, like:

- specifying the Customer in the constructor for the Order
- **`add/removeOrder`** methods associated with Customer



# Arrows: Navigability

## Conceptual

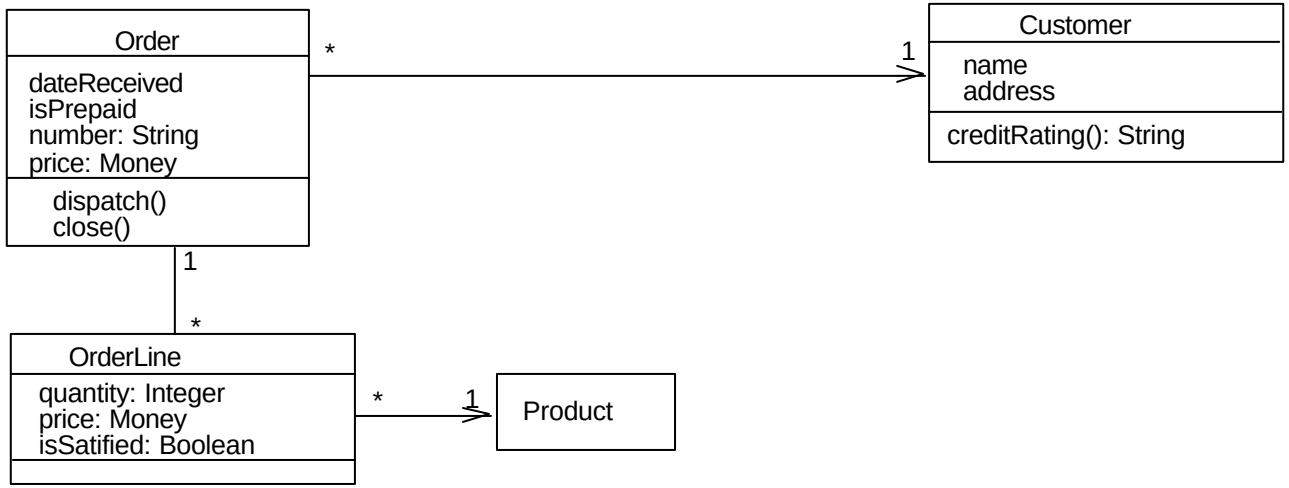
- no real sense

## Specification

- responsibility
  - an Order has the responsibility to tell which Customer it is for but Customer don't

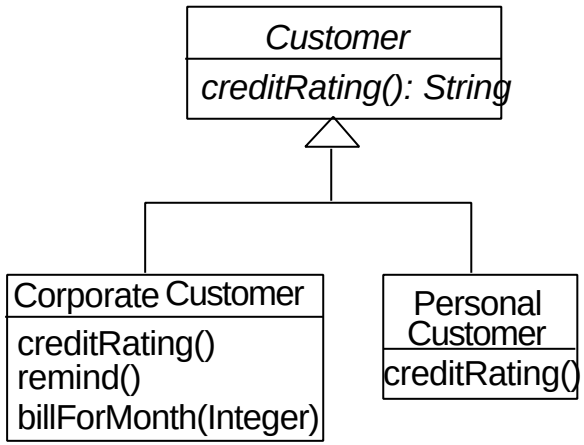
## Implementation

- depedenencies
  - an Order points to a Customer, an Customer doesn't



# Generalization

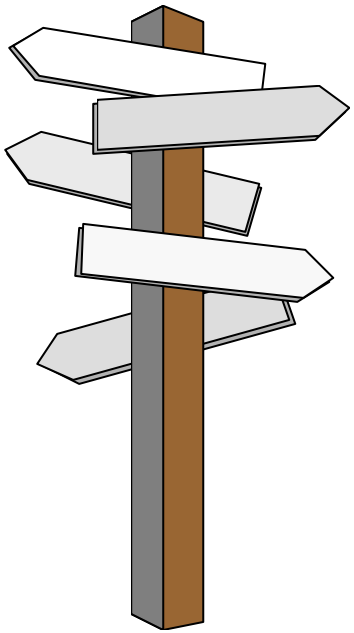
UML semantics only supports generalization and not inheritance.




- Conceptual:
  - What is true for an instance of a superclass is true for a subclass (associations, attributes, operations).
    - Corporate Customer is a Customer
- Specifications:
  - Interface of a subtype must include all elements from the interface of a superclass
- Implementation:
  - Generalization semantics is not inheritance.
  - But we should interpret it this way for representing extracted code.

# Roadmap

- Why Extracting Design? Why Uml?
- Basic Uml Static Elements
- Experimenting With Extraction
- Interpreting Uml
- **Tracks For Extraction**
- Extracting of Intention
- Extraction of Interaction
- Conclusion



# Association Extractions

Goal: Explicit references to domain classes

- Domain Objects
  - Qualify as attributes only implementation attributes that are not related to domain objects.
  - Value objects -> attributes and not associations,
  - Object by references -> associations
    - **`String name`** -- an attribute
    - **`Order order`** -- an association
    - **`Piece myPiece`** (in C++) -- composition

- Two classes possessing attributes on each other
  - an association with navigability at both side

# Language Impact on Extraction

- Attributes interpretation
- In C++
  - **`Piece* myPiece`** → aggregation or association
  - **`Piece& my Piece`** → aggregation or association
  - **`Piece myPiece`** (copied so not shared) → composition
- In Java
  - Aggregation and composition is not easy to extract
    - **`Piece myPiece`** → attribute or association or aggregation

# Method Signature for Extracting Relation

- Having attributes is not always necessary to interact with an object
  - **temporary references** exist:
    - temporary variable, method parameter, returned value
  - An instance can be dynamically created
  - An instance can pass itself as a parameter
- Do not limit yourself to attributes, methods also contain implicit relationships

```
void putPiece (int x, int y, Piece piece)
```

    - relation between a Board and a Piece
- When should we extract an aggregation and not a relation is not clear!
  - Analyse the language semantics (by copy, by reference)

# Operation Extraction (1)

- You may not extract
  - accessor methods with the name of an attribute
  - operators, non-public methods,
  - simple instance creation methods
    - new in Smalltalk, constructor with no parameters in Java
  - methods already defined in superclass,
  - methods already defined in superclass that are not abstract
  - methods that are responsible for the initialization, printing of the objects

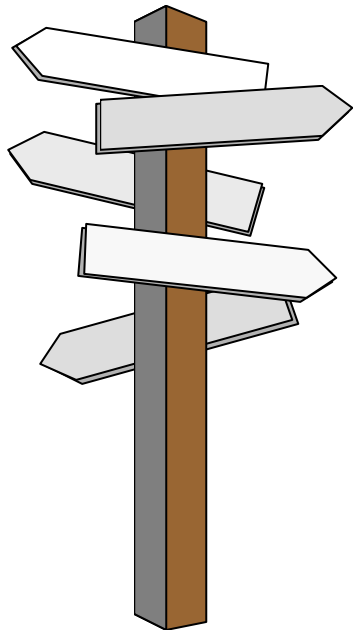
- Use company conventions to filter
  - Access to database,
  - Calls for the UI,
  - Naming patterns

# Operation Extraction (2)

- If there are several methods with more or less the same intent
  - If you want to know that the functionality exists, and not all the details
    - ⇒ select the method with the smallest prefix
  - If you want to know all the possibilities, but not all the ways you can invoke them
    - ⇒ select the method with the more parameters
- If you want to focus on important methods
  - ⇒ categorize methods according to the number of time they are referenced by clients
  - ⇒ but a hook method is not often called but still important
- What is important to show: the Creation Interface
  - Non default constructors in Java or C++

# Road map

- Why Extracting Design? Why Uml?
- Basic Uml Static Elements
- Experimenting With Extraction
- Interpreting Uml
- Tracks For Extraction
  - Extraction of Intention
- Extraction of Interaction
- Conclusion

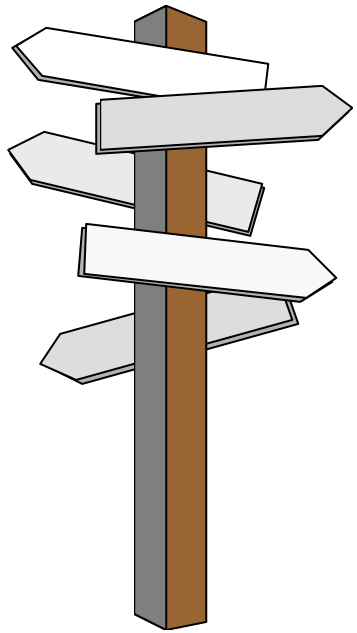


# Design Patterns as Documentation Elements

- Design Patterns reveal the intent
  - so they are definitively appealing for supporting documentation
- But...
  - Difficult to identify design patterns from the code
    - What is the difference between a State and a Strategy from the code p.o.v
  - Need somebody that knows
  - Lack of support for code annotation so difficult to keep the use of patterns and the code evolution

# Road Map

- Why Extracting Design? Why Uml?
- Basic Uml Static Elements
- Experimenting With Extraction
- Interpreting Uml
- Language Specific Issues
- Tracks For Extraction
- Extracting of Intention
- Extraction of Interactions
- Conclusion

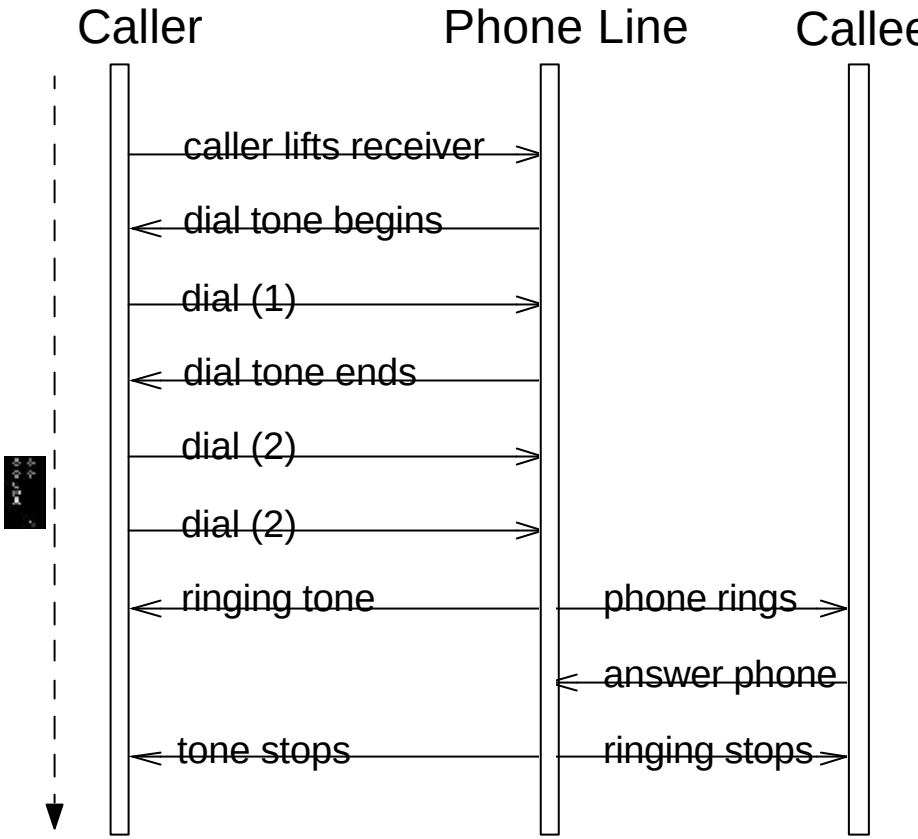


# Documenting Dynamic Behaviour

- Focusing only at static element structural elements (class, attribute, method) is limited, does not support:
  - protocols description (message A call message B)
  - describe the role that a class may play e.g., a mediator
- Calling relationships is well suited for
  - method interrelationships
  - class interrelationships
- UML proposes Interaction Diagrams
  - Sequence Diagram or Collaboration Diagram

# Sequence Diagrams

- A sequence diagram depicts a scenario by showing the interactions among a set of objects in temporal order.
- Objects (not classes!) are shown as vertical bars.
- Events or message dispatches are shown as horizontal (or slanted) arrows from the send to the receiver.
- Recall that a scenario describes a typical example of a use case, so conditionality is not expressed!



# Statically Extracting Interactions

- Pros:
  - Limited resources needed
  - Do not require code instrumentation
- Cons:
  - Need a good understanding of the system
    - state of the objects for conditional
    - compilation state #ifdef...
    - dynamic creation of objects
- Potential behavior not the real behavior
  - Blur important scenario

# Dynamically Extracting Interactions

- Pros:
  - Help to focus on a specific scenario
  - Can be applied without deep understanding of the system
- Cons:
  - Need reflective language support
    - MOP, message passing control or code instrumentation (heavy)
  - Storing retrieved information
    - may be huge
- For dealing with the huge amount of information
  - selection of the parts of the system that should be extracted, selection of the functionality
  - selection of the use cases
  - filters should be defined
    - several classes as the same, several instance as the same...
- A simple approach:
  - open a special debugger that generates specific traces

# Lessons Learnt

- You should be clear about:
  - Your goal (detailed or architectural design)
  - Conventions like navigability,
  - Language mapping based on stereotypes
  - Level of interpretations
- For Future Development
  - Emphasize literate programming approach
  - Extract design to keep it synchronized
- UML as Support for Design Extraction
  - Often fuzzy
  - Composition aggregation limited
  - Do not support well reflexive models
  - But UML is extensible, define your own stereotype