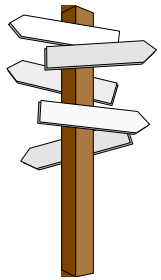
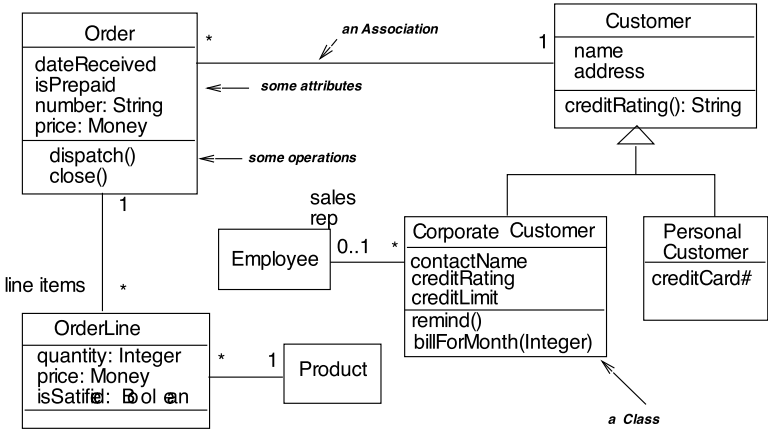
Outline

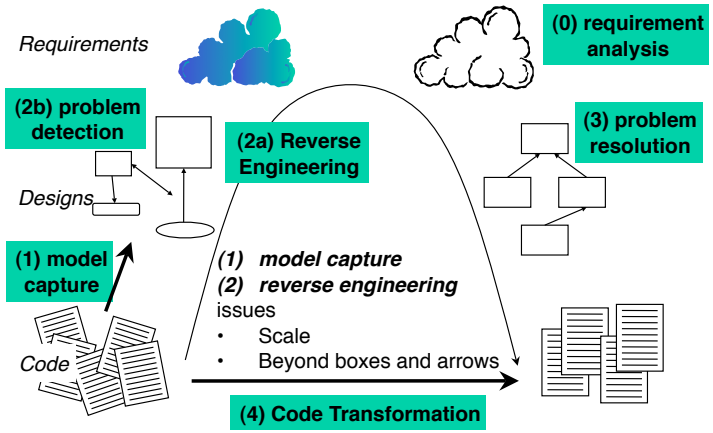
- **Why Extracting Design? Why Uml?**
- Basic UML Static Elements
- Interpreting UML
- Language Specific Issues
- Tracks For Extraction
- Extraction Techniques
- Conclusion



The Little Static UML



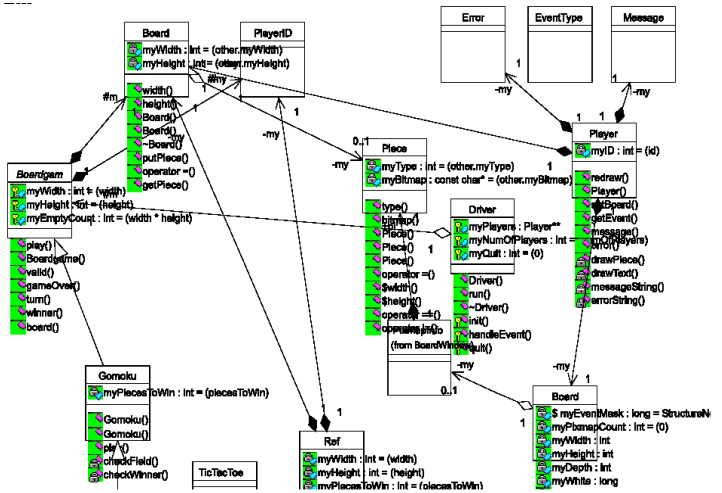
The Reengineering Life-Cycle



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 requirements

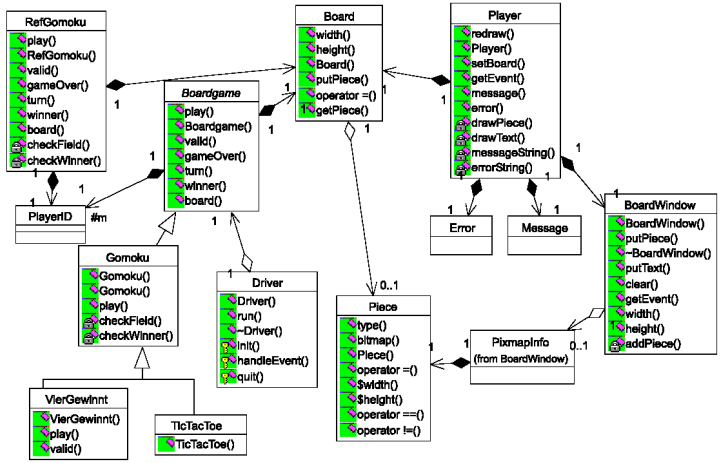
Let's Practice



Evaluation

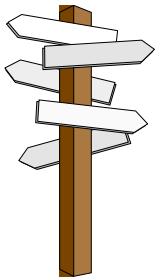
- We should have heuristics to extract the design.
- Try to clean the previous solution you found
- Try some heuristics like **removing**:
  - private information,
  - remove association with non domain entities,
  - simple constructors,
  - destructors, operators

A Cleaner View



Roadmap

- Why Extracting Design? Why Uml?
- Basic Uml Static Elements
- **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 Interpretation: Perspectives

- Fowler proposed 3 levels of interpretations called perspectives:
  - 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 Perspective

- 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: Clients cannot change it

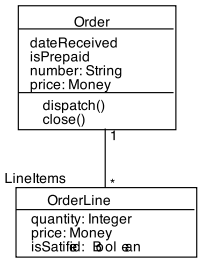
Operations in Perspective

- 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

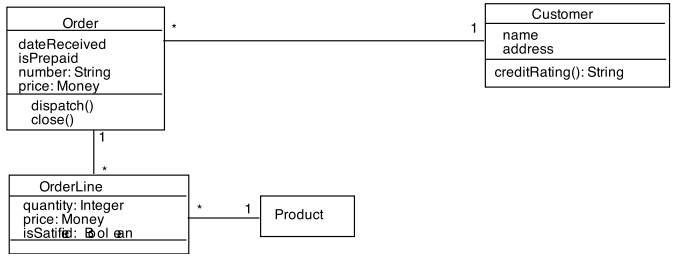
- Each association has two roles: each role is a direction on the association.
- a role can be explicitly named, labeled near the target class
- if not named from the target class and goes from a source class to a target class
- a role has a multiplicity: 1, 0, 1..*, 4
- LineItems =
  - role of direction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

Associations: Implementation Perspective

Different ways to implement an association

- class attribute, local variable, parameters
- collections

Implementation of **aggregation** and **composition**

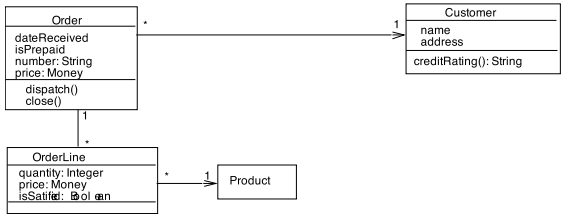
- part-of relationship
- composition – contained objects are deleted/copied when parent object is deleted/copied

Arrows: Navigability

Conceptual
- no real sense

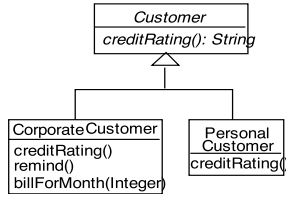
Specification
- responsibility
  - an Order has the responsibility to tell which Customer it is for but Customer don't

Implementation
- dependencies
  - an Order points to a Customer, an Customer doesn't



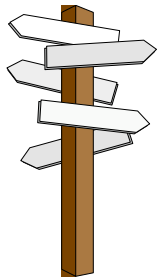
Generalization

- **Conceptual**:
  - What is true for an instance of a supertype is true for a subtype.
    - 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
- Distinguish **associations** from **attributes**
  - 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 ends

Association Extraction: Language Impact

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

Operation Extraction (1)

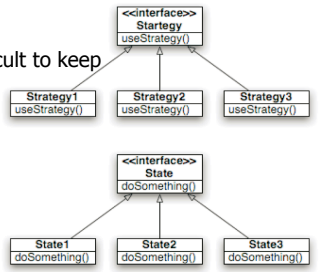
- You may **not extract**
  - accessor methods with the name of an attribute
  - operators, non-public methods,
  - simple instance creation methods
    - constructor with no parameters in Java
  - methods already defined in superclass,
    - they are inherited
  - methods that are responsible for the 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
  - ⇒ Counterexample: 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++

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

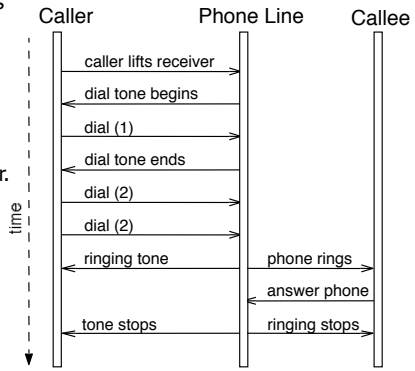


Documenting Dynamic Behaviour

- 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
    - 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