

Object-oriented Design

1. From analysis to design
2. System design
3. Object design
4. Abstraction mechanisms
5. Collaboration-based design

1. From Analysis To Design

- Entire system as an object
- Physical partitioning onto machines
- Analysis classes → design classes++
- Relationships/constraints are the key

Intermodel Consistency

- Use cases \leftrightarrow collaborations
- Statecharts for classes
- Static model methods \leftrightarrow statechart events, actions and activities
 - Internal statechart events issues by methods
 - Signals

2. System Design

- Architecture; architectural styles
- Concurrency (default: one thread / object)
- Physical design (allocate tasks to processors)
- Data stores (database vs. files; locks; protocols)
- Control (external / internal)
- Boundary conditions (e.g. handling failures)
- Trade-off priorities

3. Object Design

- a. Methods
- b. New classes
- c. Implementing relationships
- d. Misuses of inheritance
- e. Implementing control
- f. Refactoring
- g. Abstract classes, interfaces, types
- h. Information hiding (visibility)

a. Methods

- Common methods
 - attribute get / put; constructor; destructor; copy (deep / shallow); print; selection / qualification / iteration for collection classes
- Decide on operations
 - Static object model / methods / signals
 - Dynamic models: actions / activities / events
 - Assign operations to classes

b. New Classes

- For intermediate results
- To implement relationships
- For abstraction (abstract classes)
- Interfaces

c. Kinds Of Relationships

- Generalization
 - Subclasses
 - Enumeration typed attributes
- Association
 - Aggregation
 - Composition
 - General structural relationship
- Dependency (*uses*)
 - Method or constructor call
 - Attribute or global object
 - Argument passing; return value
 - Passing **this**

d. Misusing Inheritance

- Inheritance is an implementation technique
 - *Subclassing*
 - Bad: Use of inheritance to access a method or to implement PART-OF
- Generalization / specialization is an abstraction
 - *Subtyping*: all instances of the derived class are also instances of the parent class
- Extension vs. restriction
 - Is a square a rectangle in which both sides are equal?
 - Or is a rectangle a square with an additional attribute?

Principle Of Substitutability

Liskov

- *If S is a subtype of T, then objects of type T in a program may be substituted with objects of type S, without altering any of the desirable properties of that program* – Wikipedia
- Child has all visible attributes, operations, invariants, relationships and behaviors as parent
- Question: consider vegetarians and their complement (people who eat vegetables and meat). Which should be the parent class and which the child?

Implementing Associations

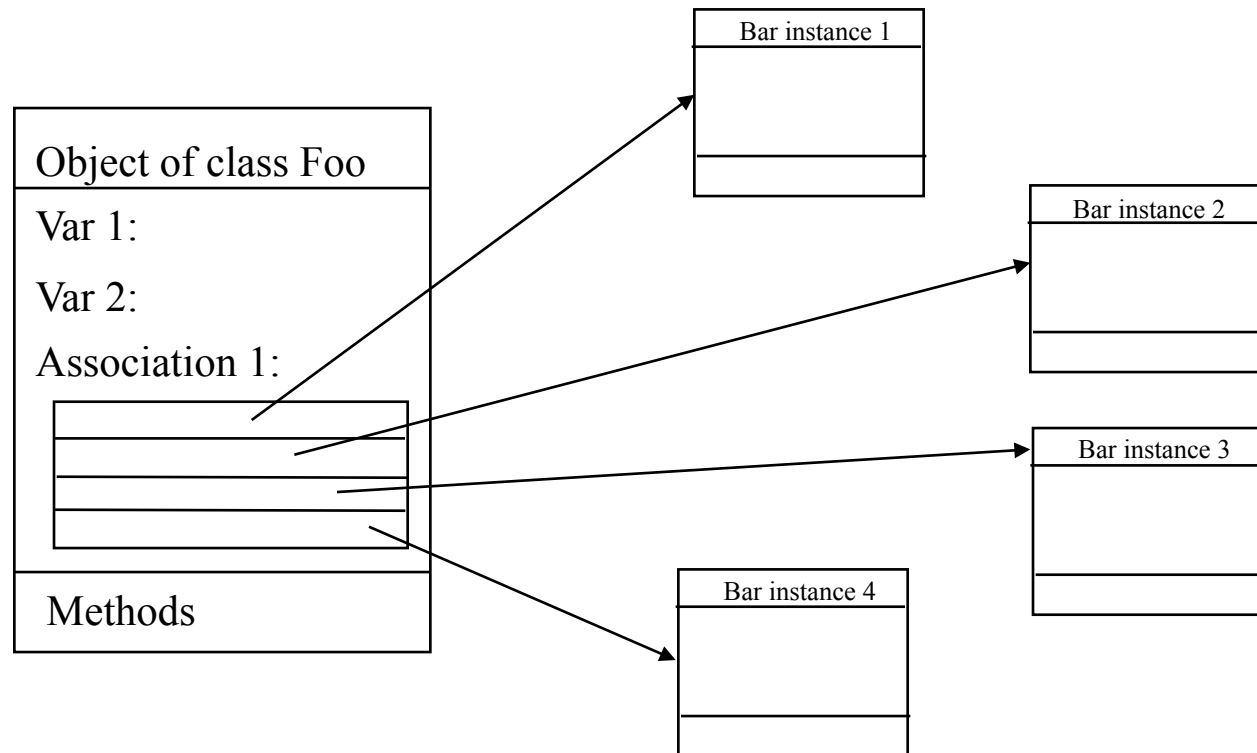
- Factors to take into account
 - Directionality
 - Cardinality
 - Access (CRUD): create, read, update, delete
- Invariant maintenance (referential integrity)

Analyze Association Traversals

- One way, 1-1: use simple pointer
- One way, 1-n: use vector of pointers
- Two way
 - Like one way but with backward search; use if one direction is much more prevalent than the other (expensive)
 - Two one ways; use if accesses outnumber updates (referential integrity problems)
 - Associations as objects

Pointers

- One directional (functional): use pointers
 - Example: `Foo` associated with `Bar`
 - One-to-one or one-to-many, one way navigation



Associations As Objects

- Implement an association between classes `Foo` and `Bar` by introducing a new (association) class: `Foo_Bar_Association` having two reference attributes:
 - `foo_instance`
 - `bar_instance`
- One instance of the association class (link) for each pair of `FOOs` and `BARs` that are associated
- `Foo` and `Bar` still have to access the association

Example

FooObject i_1	BarObject j_1
-----------------	-----------------

FooObject i_2	BarObject j_2
-----------------	-----------------

FooObject i_n	BarObject j_n
-----------------	-----------------

Tables For Associations

- Use a collection class (hash table, set, array) to hold all pairs of associated objects

FooObject 6	BarObject 7
FooObject 2	BarObject 3
FooObject 10	BarObject 6
FooObject 6	BarObject 4
FooObject 6	BarObject 3
FooObject 7	BarObject 6
FooObject 3	BarObject 2
FooObject 7	BarObject 1

Association Maintenance

- Associations express a structural relationship between instances of the incident classes
- When an instance changes it may become necessary to adjust the implementations of the constituent links of the association
 - For example, in a marriage relationship, if one spouse dies, the link between the two is broken. This might mean deleting a pointer in the spouse or removing a link entry from a table
- These kind of situations are called *referential integrity* constraints in the database world
- Failure to handle them is a notorious source of bugs

OCL Invariants

- OCL invariants, particularly those that reference classes outside their contexts, also require maintenance
- If an attribute occurring on one side of an equality changes value, it becomes necessary to change one or more values occurring on the other side in order to reestablish the invariant
- If those dependent variables denote attributes in non-context classes, then the corresponding class implementation is responsible for the update
- A variety of invariant maintenance strategies are available to the designer

e. Implementing Control

- *Ad hoc*
- State machines
 - Program variables
 - Tailored code
 - Table-driven interpreter

f. Refactoring

- *Factoring*: moving common features to the parent class
 - Reduces redundancy
 - Enforces a design constraint
- *Refactoring*: moving features around an existing class hierarchy in order to improve elegance and reuse

g. Abstract Classes

- Syntactic mechanisms for enforcing design constraints
 - *Abstract method*: a method signature without an associated implementation
 - *Abstract class*: a class containing one or more abstract methods
 - Can have no direct instances
 - Enforces a design constraint on child classes
 - *Interface*: an abstract class in which all methods are abstract (e. g. `serializable`) and in which there are no non-final attributes
 - *Type*: interface with attributes

h. Information Hiding

- Improve maintainability by hiding representation decisions behind abstract interfaces
- Word processing application variants
 - No hiding/published representation (e.g. `char *`)
 - Collection of code in a file (possibly `static`)
 - `WORD` data type with access routines
 - Packages / namespaces
 - Protected
 - Private

4. Abstraction Mechanisms

Mechanism	Example
Clichés / idioms	<pre>while (*t) { *s++=*t++; } *s='\0';</pre>
Classes	hashtable
Patterns	visitor
Aspects	logging
Frameworks	model-view-controller
Architectural styles	client-server

5. Collaboration-Based Design

- User stories
- CRC cards
- Collaboration and role identification
- Documentation via sequence or collaboration diagrams
- Role abstraction via abstract classes / interfaces
- Synthesis of classes from roles