Object-Oriented Design

2501ICT Nathan

René Hexel

School of Information and Communication Technology
Griffith University

Semester 1, 2011

Outline

Object-Oriented Design

Object-Oriented Paradigms and Design

History and Motivation

- In the 60s, programs started to become complex
- Modules
  - sub-divide code into multiple modules
  - each module can contain several, related functions

  → problem: how to share data between these functions?

  → Without Global Variables!!!
- Simula (1967)
  - programming language for performing simulations
  - idea: group different types of objects into classes

  → each class is responsible for its own data and behaviour

  → multiple instances of the same class can exist!
- Smalltalk (1969)
  - introduction of the term *Object Oriented*

  → objects are independent of each other

  → messages are used to communicate between objects

Basics of Object Orientation

- Classes
  - abstract definition of things (objects), including
  - → the object's attributes (properties), and
  - → the object's behaviour (methods)
- Objects (Instances)
  - concrete instantiations of classes, e.g.:
  - → multiple individuals of the class `Student`
- Inheritance: Subclasses
  - a subclass is a more specialised versions of a class
  - → inherits all attributes and behaviours of their parent class
  - → can introduce their own, new attributes and behaviours

Basics of Object Orientation (2)

- Methods
  - the ability of objects to perform an operation, e.g.:
  - → a `name` method that returns a string representing the `Student`'s name
- Message passing
  - the process by which one object asks another object to invoke a method
  - → usually, a method corresponding to the message name is invoked
  - → can be delegated to other methods or even other objects!
- Information Hiding and Encapsulation
  - reveal as little information about classes as necessary
  - → no-one on the outside can rely on hidden information
  - → only hidden information can be changed later

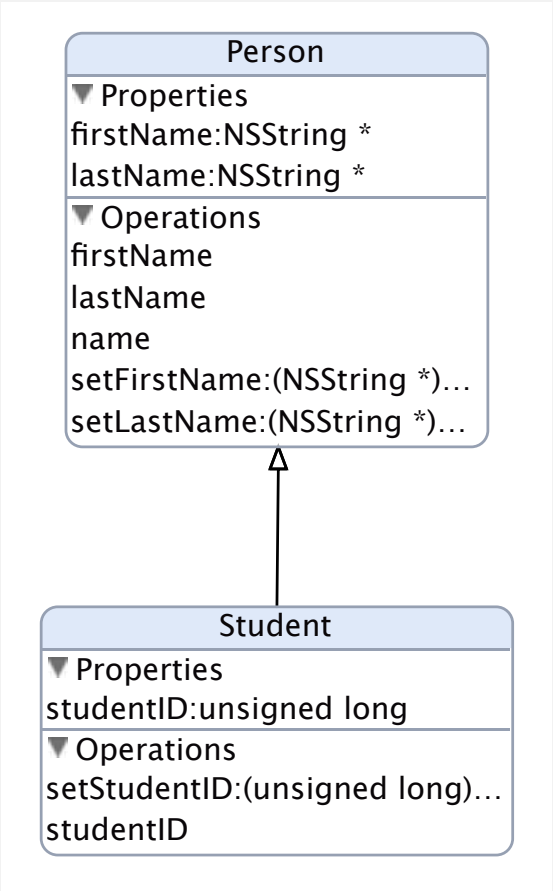
Object Oriented Design

- Divide and Conquer
  - repeatedly sub-divide a large problem into a hierarchy of smaller problems
  - → result: a large number of very small (and easy to solve) sub-problems
  - → solving each sub-problem is the responsibility of a class
- Responsibility-driven Design
  - What actions is a class responsible for?
  - What information does this class represent?
- Data-driven Design
  - What Abstract Data Types are required?
  - What concrete implementation should be used?

High Level Design

- Think about the problem first!
  - What classes are required?
  - How should they interact?
- Class Diagrams
  - describe the structure of a system (program)
  - show the attributes and relationships between classes
- UML (Unified Modeling Language)
  - general-purpose modeling language for software engineering
  - allows modelling using diagrams

UML Diagram Example



Design Patterns

- Design Pattern Definition
  - recurring solution to a common problem in software design
  - description of how to solve a problem that appears in many different situations

  → documents a solution to a particular design problem

  → scheme that can be reused over and over again
- Advantages
  - speed up the development process
  - provide tested solutions and proven development paradigms
- Challenges
  - creating flexible, universal design patterns is complex
  - often evolve over time as understanding grows

Iterator Design Pattern

- Problem: provide a way of accessing collection elements without exposing an underlying representation
  - → e.g.: how can a linear collection be enumerated if I don't know if it is implemented as an array or a linked list?
- E.g.: `NSEnumerator`
  - abstract class that allows traversing collections
  - concrete subclass comes from the collection class implementation
  - Examples
    - `NSArray -objectEnumerator` method
    - `NSDictionary -keyEnumerator` method

Delegation Design Pattern

- An object that forwards a task to a different object instead of performing the task itself.

Example

```
@interface SomeClass: NSObject
{
  id delegate;
}
- (void) finishedProcessing;
@end

@implementation SomeClass
- (void) finishedProcessing
{
  if ([delegate respondsToSelector:
          @selector(finishedProcessing)])
  {
    [delegate finishedProcessing];
  }
}
@end
```

Target-Action Mechanism

- Extension of the delegation design pattern
  - → instead of registering a single delegate, a *target object* registers its interest in a particular action
    - allows multiple targets to be registered for different events
      - e.g. a GUI Button Widget allows separate registration for when the button was pressed and when it was released
- Target
  - → object to be notified
- Action
  - → method that should be invoked (method selector)

Mediator Design Pattern

- A Class that encapsulates how objects interact
  - → promotes loose coupling
  - → objects do not need to know anything about each other
  - → (only the mediator needs to know about the objects)
- Model-View-Controller (MVC) paradigm
  - GUI Design: data models should be independent of the user interface
  - Problem: how to connect different GUIs to underlying models?
  - → Controller
    - sits in the middle between the Model and the View
    - → mediates between the two
    - → model does not need to know anything about the view and vice versa

Factory Methods

- Problem: creating objects without specifying the exact class of object that will be created
  - → subclasses can override the factory method
- Class Cluster
  - Abstract class factory method instatiates a concrete subclass
- Examples
  - `NSString *s = [NSString stringWithUTF8String: "foo"];`
  - `NSString *t = [NSString stringWithContentsOfFile: @"file.txt"];`

Lazy initialisation

- Tactics of delaying the creation of an object until it is actually needed. This is particularly useful if creating such an object is very expensive (e.g. loading from disk, or downloading from the network).

Example

```
@interface SomeClass: NSObject
{
  NSMutableArray *array;
}
@end

@implementation SomeClass
- addWord: (NSString *) word
{
  if (!array)
    array = [NSMutableArray new];

  [array addObject: word];
}

- (void) dealloc
{
  [array release];
  [super dealloc];
}
@end
```