



## Grain Package

- Grain is an object-oriented interface to the windowing system.
- Grain allows you to create various kinds of windows, to draw graphics and text in those windows and to handle events.
- The Grain toolkit is very simplistic and portable, which make it easy to use.
- Grain is a proprietary toolkit only used for educational purposes but not suitable for professional applications.
- wxWindows is a cross-platform GUI toolkit based on C++ that is widely used in academia and industry



## Features of Grain

- Grain consists of only six classes
- Grain can handle
  - Colors
  - Fonts
  - Base windows
  - Sub-windows
  - Pop-up windows
  - Drawing of text, points, lines, rectangles
  - Events such as pointer motion, mouse button and keyboard events



## Grain

- Grain supports
  - Any number of *base windows* can be created
  - A base window can be divided into *sub-windows*, each with its own coordinate system and event handler
  - Temporary pop-up windows can be created for messages and short user dialogs
  - Lines and rectangular regions can be drawn in any window. Color, fill style and line width can be selected
  - Text can be drawn in different fonts and colors
  - Events occurring in a window can be detected by the application program



## Window Types

- Base windows
  - Base windows lie directly on the screen background. They can be moved and possibly resized by the user. They have a border line and a title.
- Sub-windows
  - Sub-windows represent a part of another window, which is called the parent. A subwindow will not be drawn outside the borders of its parent. It can not move itself but only together with its parent.
- Pop-up windows
  - Pop-up windows are temporary windows that are shown briefly, for example a pop-up menu.

## Classes in Grain

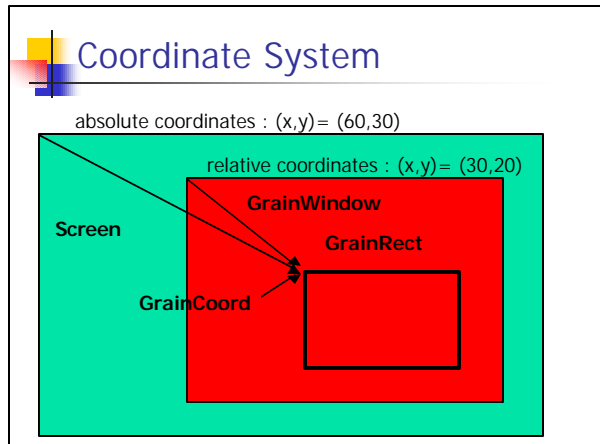
- GrainWindowingSystem**
  - There should be exactly one instance of **GrainWindowingSystem** which is used to create and manipulate windows.
  - GrainWindowingSystem** is an abstract class, from which the concrete sub-class **GrainXWindowingSystem** is derived.
- GrainWindow**
  - Captures all three types of windows and has methods for drawing text, points, lines and rectangles.
  - Each window has its own integer coordinate system with the origin in the upper left corner.

## Classes in Grain

- GrainWindowEventHandler**
  - A **GrainWindowEventHandler** is connected with a **GrainWindow** and detects events. The **GrainWindow** object has a pointer to a **GrainWindowEventHandler** object.
  - The windowing system tells the event handler when contents of the window needs to be redrawn (refresh event)
  - The **GrainWindowEventHandler** class does nothing. The application program has to derive a sub-class from **GrainWindowEventHandler** that redefines at least some of the event handling methods.

## Classes in Grain

- GrainColor**
  - A **GrainColor** objects represent colors (RGB-values) and fill-style (boolean). Some colors like transparent, red, white, green, blue, black are predefined.
- GrainCoord**
  - A **GrainCoord** object represents a position or size (width and height). It has two public integer members x and y.
  - Simple coordinate arithmetic is supported by +, -, +=, -= operators.
- GrainRect**
  - A **GrainRect** object represents a rectangle. It has two **GrainCoord** data members, position and size, position.x and position.y specify the upper-left corner, size.x and size.y represent the width and height of the rectangle



## Creating and Destroying Windows

- The **GrainWindowingSystem** object creates new base windows and pop-up windows using the methods (bounds in absolute coordinates)
  - **GrainWindow\* create\_base\_window** (const **GrainRect& bounds**, **bool resizable**, **const string& title**);
  - **GrainWindow\* create\_pop\_up\_window** (const **GrainRect& bounds**);
- The **GrainWindow** object creates new sub-windows using the method (bounds in relative coordinates)
  - **GrainWindow\* create\_subwindow** (const **GrainRect& bounds**);

## Drawing into a Window

- **GrainWindow**
  - A single point can be drawn with the method **draw\_point(const GrainCoord& at, const GrainColor& color)**;
  - A set of points can be drawn with the method **draw\_points(const vector<GrainCoord>& at, const GrainColor& color)**;
  - A line can be drawn with the method **draw\_line(const GrainCoord& from, const GrainCoord& to, const GrainColor& color, size\_t line\_width)**;

## Drawing into a Window

- **GrainWindow**
  - A rectangle can be drawn with the method **draw\_rectangle(const GrainRect& bounds, const GrainColor& border\_color, const GrainColor& fill\_color, size\_t line\_width)**;
  - Text can be drawn with the method **draw\_text(const GrainCoord& at, const string& text, const string& font\_name, const GrainColor& color)**;
  - The width and height of a string in a particular font is calculated by the method **GrainCoord GrainWindowingSystem::text\_size(const string& font\_name, const string& text)**;

## Drawing Sample Program

```
#include "grain/grain.hh"
int main()
{
    GrainXWindowingSystem system; // create WindowSystem object
    GrainWindow* window = system.create_base_window(
        GrainRect(GrainCoord(0,0), GrainCoord(200,200)), false, "Title");
    window->draw_rectangle( GrainRect(GrainCoord(20,20), GrainCoord(40,30)),
        GrainColor::black, GrainColor::red, 3); // draw a rectangle
    window->draw_text( GrainCoord(20,20), "Hello World", "9x15",
        GrainColor::black); // draw some text
    window->refresh(); // refresh window to display changes
}
```

## Grain Pointer Events

- A window has an associated event handler of the class **GrainWindowEventHandler** that uses methods to detect
  - Pointer events
    - **void press\_event(GrainWindow\* w, const GrainCoord& pos, size\_t button, const set<Modifier>& modifiers);**  
enum Modifier { Shift, Control, Meta, Alt };
    - **void release\_event(GrainWindow\* w, const GrainCoord& pos, size\_t button);**
    - **void motion\_event(GrainWindow\* w, const GrainCoord& pos);**
    - **void enter\_event(GrainWindow\* w, const GrainCoord& pos);**
    - **void leave\_event(GrainWindow\* w, const GrainCoord& pos);**

## Grain Keyboard / Refresh Events

- Keyboard event allows the event handler to detect when a key on the keyboard is pressed  
**void key\_event(GrainWindow\* w, const string& value, const set<Modifier>& modifiers );**
- Refresh event is invoked by the windowing system when the contents of the window needs to be redrawn  
**void refresh\_event(GrainWindow\* w, const GrainRect& bounds);**
- The event handler function in the base class GrainWindowEventHandler do nothing, therefore the application has to redefine at least some of the event handling methods.

## Example EventHandler

```
class EventHandler : public GrainWindowEventHandler
{
public:
    EventHandler();
    void press_event(GrainWindow* w, const GrainCoord& pos,
        size_t button, const set<Modifier>& modifiers);
    void release_event(GrainWindow* w, const GrainCoord&
        pos, size_t button);
    void refresh_event(GrainWindow* w, const GrainRect&
        bounds);
    void key_event(GrainWindow* w, const string& value,
        const set<Modifier>& modifiers);
private:
    bool pressed; // mouse status
    string s;     // stores last key pressed
};
```

## Example EventHandler

```
EventHandler::EventHandler() : pressed(false), s(" ") {}

void EventHandler::press_event(GrainWindow* w, const GrainCoord&
    pos, size_t button, const set<Modifier>& modifiers)
{
    pressed=true; // indicate mouse status pressed
    w->refresh(); // generate a refresh event
}

void EventHandler::release_event(GrainWindow* w, const
    GrainCoord& pos, size_t button)
{
    pressed=false; // indicate mouse status released
    w->refresh(); // generate a refresh event
}
```

## Example EventHandler

```
void EventHandler::refresh_event(GrainWindow* w, const
    GrainRect & bounds)
{
    if (pressed)
    {
        w->draw_rectangle(GrainRect(GrainCoord(50, 10),
            GrainCoord(20,20)),GrainColor::blue, GrainColor::red, 3);
        w->draw_text(GrainCoord(50, 10), s, "9x15",
            GrainColor::blue);
    }
    else
    {
        w->draw_rectangle( GrainRect(GrainCoord(50, 10),
            GrainCoord(20,20)),GrainColor::red, GrainColor::blue, 3);
        w->draw_text(GrainCoord(50, 10), s, "9x15",
            GrainColor::red);
    }
}
```

## Example EventHandler

```
void EventHandler::key_event(GrainWindow* w, const
    string& value, const set<Modifier>& modifiers)
{
    s=value; // set new character to display
    w->refresh();// refresh window
    if (value == "q") // exit if key "q" pressed
        w->system().exit_event_loop(); // exit event loop
}
```

## Example EventHandler

```
int main()
{
    GrainXWindowingSystem system; // create window system
    GrainWindow* window=system.create_base_window(
        GrainRect(GrainCoord(0, 0),GrainCoord(200,200)),
        false, "Title"); // create window object

    EventHandler handler; // create event handler object
    window->set_event_handler(&handler);
    // associate event handler to window
    system.enter_event_loop();
    // enter event loop

    return 0;
}
```

## Unified Modeling Language

- UML a complete **language** for capturing knowledge (semantics) about a subject and expressing knowledge (syntax) regarding the subject.
- **Modeling** involves a focus on understanding (knowing) a subject (system) and capturing and being able to communicate this knowledge.
- UML is the result of **unifying** the information systems and technology industry's best engineering practices (principles, techniques, methods and tools)

## Unified Modeling Language

- UML is used for specification, visualization and documentation of systems
- UML is based on the object oriented paradigm
- UML applies to a multitude of different types of systems, domains and methods or processes.
- UML unifies the syntax and semantics of methods and tools previously used for object oriented design
- UML was originally conceived by Rational Software Corporation and the "Three Amigos" in 1995
  - Grady Booch
  - James Rumbaugh
  - Ivar Jacobson

## Unified Modeling Language

- Use case diagrams describe the functionality of a system and users of the system. They contain the following elements
  - Actors which represent users of the system, including human users and other systems
  - Use cases which represent functionality or services provided by a system to users.

## Unified Modeling Language

- UML object model describes the static structure of the problem domain, it contains
  - Class diagram describes the class attributes (name and type), operations and the relationship among classes. A class diagram contains classes and associations
  - Object diagram A class model describes all possible situation, whereas an object model describes a particular situation. An object diagram contains objects and links which represent the relationship between objects

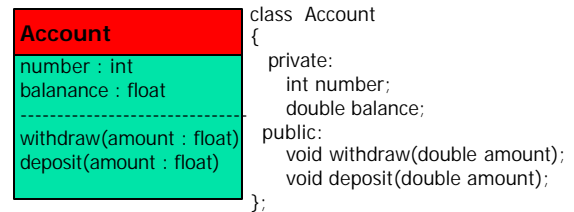
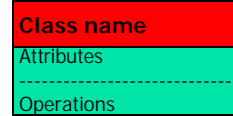
## Unified Modeling Language

- The UML dynamic model describes all temporal and dynamic aspects of the system, for example interaction among objects it contains
  - Sequence diagrams describe interactions among classes which are modeled as exchanges of messages. Sequence diagrams contain class roles, lifelines, activations and messages
  - Scenarios describe interactions among objects
  - Statechart diagrams describe the states and responses of a class. They contain states and transitions.

## Class Diagrams

- Class diagrams describe the static structure of a system rather than its behaviour. Class diagrams contain the following elements:
  - Classes
    - Which represent entities with common characteristics or features. These features include attributes (data members), operations (member functions) and associations
  - Associations
    - which represent relationship that relate two or more other classes where the relationships have common characteristics or features.

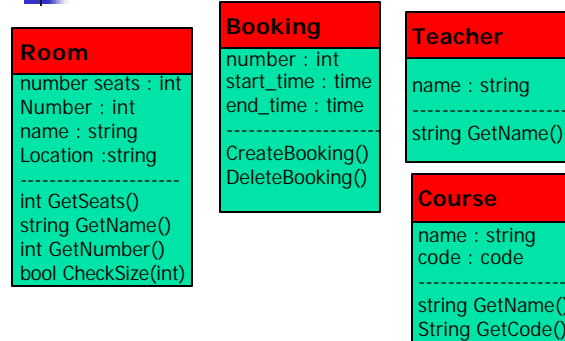
## Class Diagram

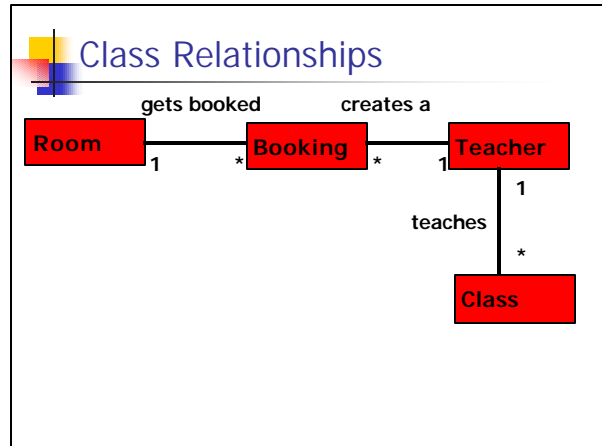
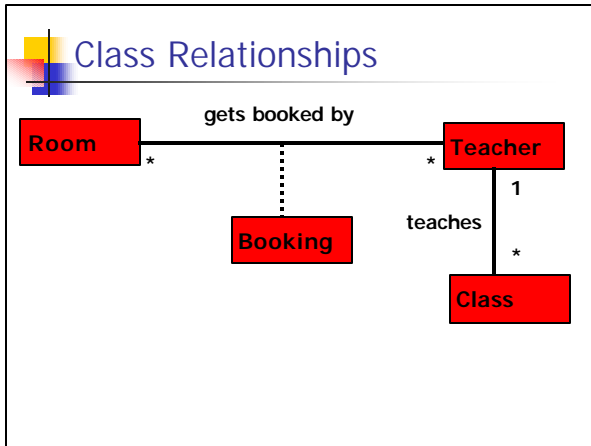


## Room Booking System

- Develop a room reservation system that allows teachers to book lecture halls for their classes.
- Each reservation contains information about the room that is booked, the date, start and end time, class, the name of the person who booked the room and a unique booking number.
- The reservation system maintains a list of lecture halls including their features such as name, location, size and number.

## Class Diagrams

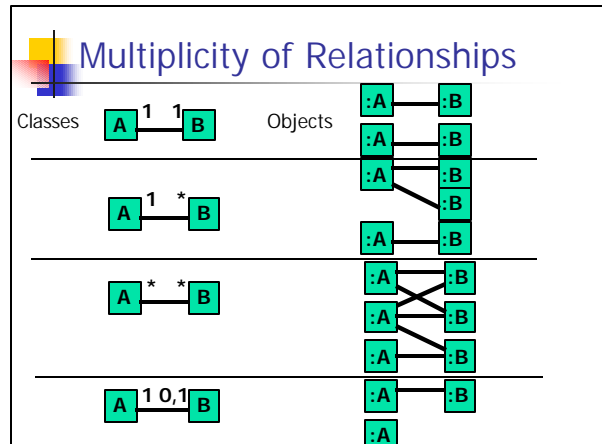




### Multiplicity of Class Relations

Multiplicity denotes how many objects of one class are associated with how many objects of the other class. The following symbols can be used to denote the multiplicity of a relationship

- 1 : exactly one object is involved in the relationship
- \* : some objects are involved in the relationship
- 1..\* : some objects but at least one
- 0..\* : some objects but possibly zero
- 0,1 : zero or one object



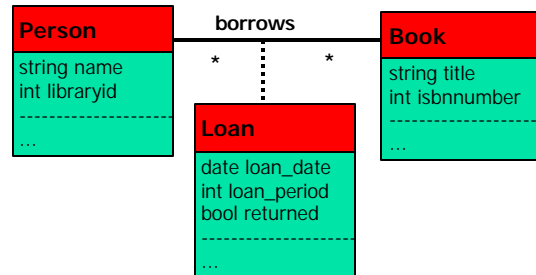
## Association

One object uses another to help it carry out a task. Classes that collaborate are usually related through associations. This type of relationship is also called a *uses* relationship.

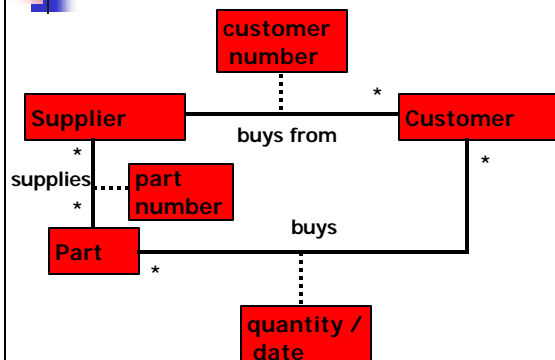


## Association Class

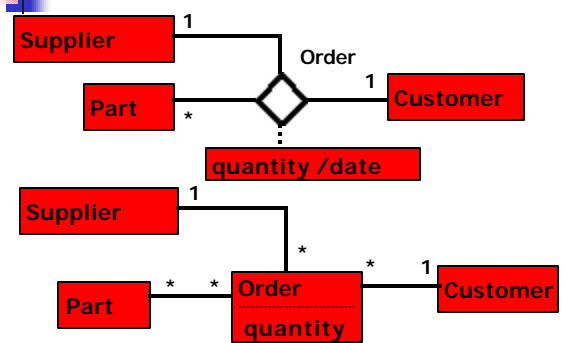
- An association class is a class that contains information about the relationship among other associated classes.



## Association Classes

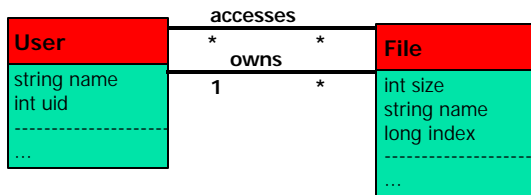


## Higher Order Associations



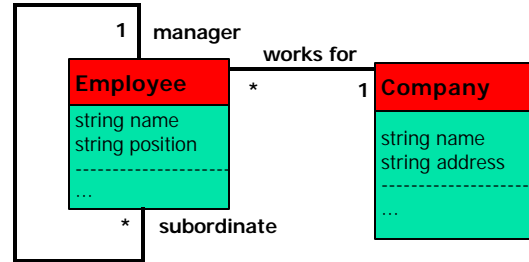
## Multiple Associations

- Sometimes two classes are associated with each other in more than one way



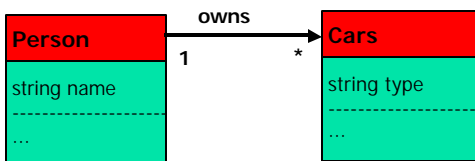
## Self-Associations

- Sometimes a class is associated to itself as different objects play different roles



## Navigability

- An association can indicate the responsibility of the involved objects by an arrow that indicates the direction of the association. If navigability exists only in one direction we call the association uni-directional otherwise bidirectional.



## Aggregation

Aggregation means that one object contains other objects. Aggregation is also called part-of relationship.



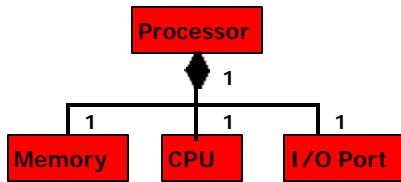
```

class Person
{ string name; ...}

class Addressbook
{ vector <Person*> persons; ... }
  
```

## Composition

Composition is building objects from parts. It is strong type of aggregation in which the parts are necessary to the whole, namely they are permanently bound to the object and do not exist outside the object. Composition is "is build of" relationship



## Composition / Aggregation

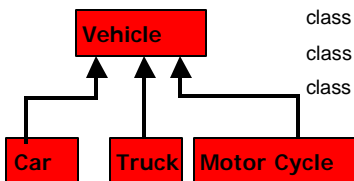


```

class Line
{ string text; ...; }
class Paragraph
{ vector <Line> lines; ...; }
class Figure { ...; }
class Document
{ vector <Paragraph> paragraphs;
  vector <Figure*> figures; }
    
```

## Generalization

Generalization is a relationship defined at the class level not the object level, which means that all objects of this class must obey the relationship. This is type of relationship is also called a *is-a-kind-of* relationship.



```

class Vehicle:
class Car : public Vehicle
class Truck : public Vehicle
class MotorCycle : public Vehic
    
```