PUC RIO

Universidade Federal Fluminense

Computer Graphics
for Engineering

TU Graz
Graz University of Technology

ifb

numsim
Numerical simulation in technical sciences

Object Oriented Modeling

Luiz Fernando Martha
André Pereira

Graz, Austria
June 2014

Outline

- Basic Object-Oriented Concepts
- UML (*Unified Modeling Language*)
- Object-Oriented Software Modeling:
  RPN Calculator
- Introduction to Design Patterns

Object-Oriented Approach

Object-Orientation

Most of the methods used in software development houses are based on a **functional** and/or **data-driven decomposition** of the systems. These approaches differ in many ways from the approaches taken by **object-oriented methods** where **data and functions are highly integrated**.

Object-oriented systems development is a way to develop software by building **self-contained modules or objects** that can be **easily replaced, modified and reused**. It depicts the view of real world as a system of cooperative and collaborative objects. In this, software is **a collection of discrete objects that encapsulates data and operations** performed on that data to model real world objects**.** A **class** describes a group of objects having similar structures and similar operations.

Object Oriented Philosophy is **very much similar to real world** and hence is gaining popularity as the systems here are seen as a set of interacting objects as in the real world. To implement this concept, the process-based structural programming is not used; instead **objects are created using data structures**. Just as every programming language provides various data types and various variables of that type can be created, similarly, in case of objects certain data types are predefined. (Nath, 2014 – Lecture Notes on Object-Oriented Methodology)

Object-Orientation

The object-oriented approach allows better organization, versatility and reuse of source code, which facilitates upgrades and improvements in the programs. The object-oriented approach is characterized by the use of classes and objects, and other concepts that will be clarified below.

• **Classes** are kinds of automakers objects that define its characteristics such as which functions are capable of performing and which attributes the object has. This type of programming allows the user to solve problems using real world concepts.

• **Object** is an instance generated from a class. An object is identified from the methods and attributes it has.

• **Encapsulation** is the act of hiding from the users the internal processes of an object, class or method.

• **Inheritance (and polymorphism)** is a feature that allows a given class to inherit the characteristics of another class. That is, the descendant acquire all methods and attributes of the parent class.

Methods are functions that object can perform.
Attribute is everything an object has as a variable.

Object-Orientation

Class, Object and Encapsulation

```
#ifndef STACK_H
#define STACK_H



class Stack
{
public:
  Stack();
  ~Stack();
  void push(double _n);
  double pop();
  bool isEmpty();
  void show();

private:
  int m_top;
  double *m_elems;
};

#endif
```

```
#ifndef REAL_H
#define REAL_H



class Real
{
public:
  Real(double _val);
  ~Real();
  Real sum(Real _n);
  Real sub(Real _n);
  Real mul(Real _n);
  Real div(Real _n);

private:
  double m_value;
};


#endif
```

Object-Orientation

Class, Object and Encapsulation

```
#ifndef STACK_H
#define STACK_H



class Stack
{
public:
  Stack();
  ~Stack();
  void push(double _n);
  double pop();
  bool isEmpty();
  void show();

private:
  int m_top;
  double *m_elems;
};

#endif
```

```
#ifndef STACK_H
#define STACK_H

#include "real.h"

class Stack
{
public:
  Stack();
  ~Stack();
  void push(Real _n);
  Real pop();
  bool isEmpty();
  void show();

private:
  int m_top;
  Real* m_elems;
};

#endif
```

```
#ifndef REAL_H
#define REAL_H



class Real
{
public:
  Real(double _val);
  ~Real();
  Real sum(Real _n);
  Real sub(Real _n);
  Real mul(Real _n);
  Real div(Real _n);

private:
  double m_value;
};


#endif
```

Object-Orientation

The object-oriented approach allows better organization, versatility and reuse of source code, which facilitates upgrades and improvements in the programs. The object-oriented approach is characterized by the use of classes and objects, and other concepts that will be clarified below.

• **Classes** are kinds of automakers objects that define its characteristics such as which functions are capable of performing and which attributes the object has. This type of programming allows the user to solve problems using real world concepts.

• **Object** is an instance generated from a class. An object is identified from the methods and attributes it has.

• **Encapsulation** is the act of hiding from the users the internal processes of an object, class or method.

• **Inheritance (and polymorphism)** is a feature that allows a given class to inherit the characteristics of another class. That is, the descendant acquire all methods and attributes of the parent class.

Methods are functions that object can perform.
Attribute is everything an object has as a variable.

Object-Orientation

Inheritance and Polymorphism

```
#ifndef REAL_H
#define REAL_H

class Real
{
public:
  Real(double _val);
  ~Real();
  Real sum(Real _n);
  Real sub(Real _n);
  Real mul(Real _n);
  Real div(Real _n);

private:
  double m_value;
};

#endif
```

```
#ifndef COMPLEX_H
#define COMPLEX_H

class Complex
{
public:
  Complex(double _re,
          double _im);
  ~Complex();
  Complex sum(Complex _n);
  Complex sub(Complex _n);
  Complex mul(Complex _n);
  Complex div(Complex _n);

private:
  double m_real;
  double m_imag;
};

#endif
```

Object-Orientation

Inheritance and Polymorphism

```
#ifndef REAL_H
#define REAL_H

class Real
{
public:
  Real(double _val);
  ~Real();
  Real sum(Real _n);
  Real sub(Real _n);
  Real mul(Real _n);
  Real div(Real _n);

private:
  double m_value;
};

#endif
```

```
#ifndef COMPLEX_H
#define COMPLEX_H

class Complex
{
public:
  Complex(double _re,
          double _im);
  ~Complex();
  Complex sum(Complex _n);
  Complex sub(Complex _n);
  Complex mul(Complex _n);
  Complex div(Complex _n);

private:
  double m_real;
  double m_imag;
};

#endif
```

```
#ifndef NUMBER_H
#define NUMBER_H

class Number
{
public:
  Number(int _type);
  ~Number();
  Number sum(Number _n);
  Number sub(Number _n);
  Number mul(Number _n);
  Number div(Number _n);

private:
  int m_type;
};

#endif
```

Object-Orientation

Inheritance and Polymorphism

```
#ifndef REAL_H
#define REAL_H

#include "number.h"

class Real : Number
{
public:
  Real(double _val);
  ~Real();
  Number sum(Number _n);
  Number sub(Number _n);
  Number mul(Number _n);
  Number div(Number _n);

private:
  double m_value;
};

#endif
```

```
#ifndef COMPLEX_H
#define COMPLEX_H

#include "number.h"

class Complex : Number
{
public:
  Complex(double _re,
          double _im);
  ~Complex();
  Number sum(Number _n);
  Number sub(Number _n);
  Number mul(Number _n);
  Number div(Number _n);

private:
  double m_real;
  double m_imag;
};

#endif
```

```
#ifndef NUMBER_H
#define NUMBER_H



class Number
{
public:
  Number(int _type);
  ~Number();
  Number sum(Number _n);
  Number sub(Number _n);
  Number mul(Number _n);
  Number div(Number _n);

protected:
  int m_type;
};

#endif
```

Object-Orientation

Inheritance and Polymorphism

```
#ifndef REAL_H
#define REAL_H

#include "number.h"

class Real : Number
{
public:
  Real(double _val);
  ~Real();
  Number sum(Number _n);
  Number sub(Number _n);
  Number mul(Number _n);
  Number div(Number _n);

private:
  double m_value;
};

#endif
```

```
#ifndef COMPLEX_H
#define COMPLEX_H

#include "number.h"

class Complex : Number
{
public:
  Complex(double _re,
          double _im);
  ~Complex();
  Number sum(Number _n);
  Number sub(Number _n);
  Number mul(Number _n);
  Number div(Number _n);

private:
  double m_real;
  double m_imag;
};

#endif
```

```
#ifndef NUMBER_H
#define NUMBER_H



class Number
{
public:
  Number(int _type);
  ~Number();
  virtual Number sum(Number _n) = 0;
  virtual Number sub(Number _n) = 0;
  virtual Number mul(Number _n) = 0;
  virtual Number div(Number _n) = 0;

protected:
  int m_type;
};

#endif
```

UML
Unified Modeling Language

UML

Unified Modeling Language

UML is an industry-standard language for:

Specifying

Visualizing

Constructing

Documenting

Business Modeling

Communications

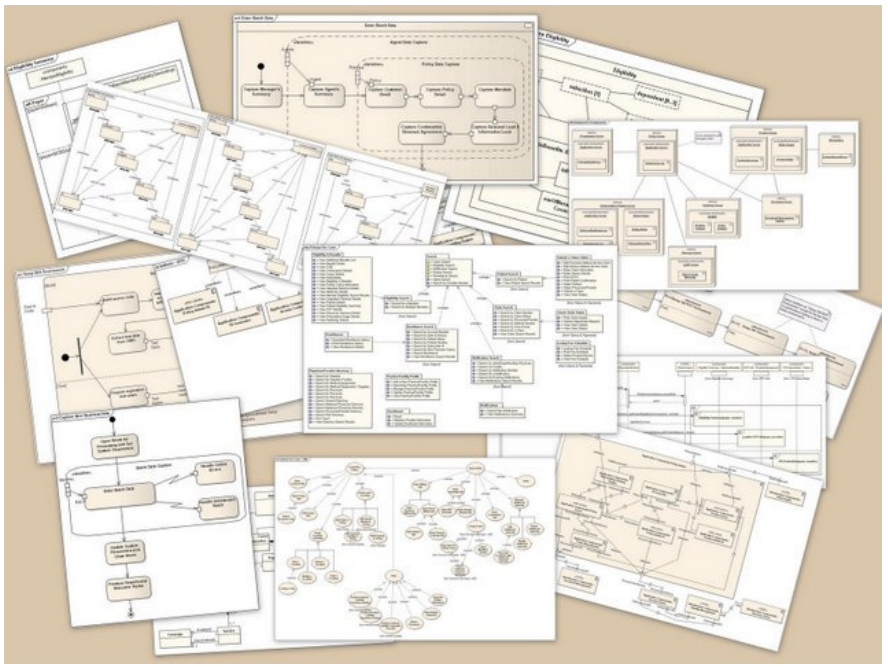
Software Components

UML

Unified Modeling Language

Definition:

It is a graphical language for visualizing, specifying, constructing and documenting the artifacts of an object-oriented computing system.



UML

Unified Modeling Language

Definition:

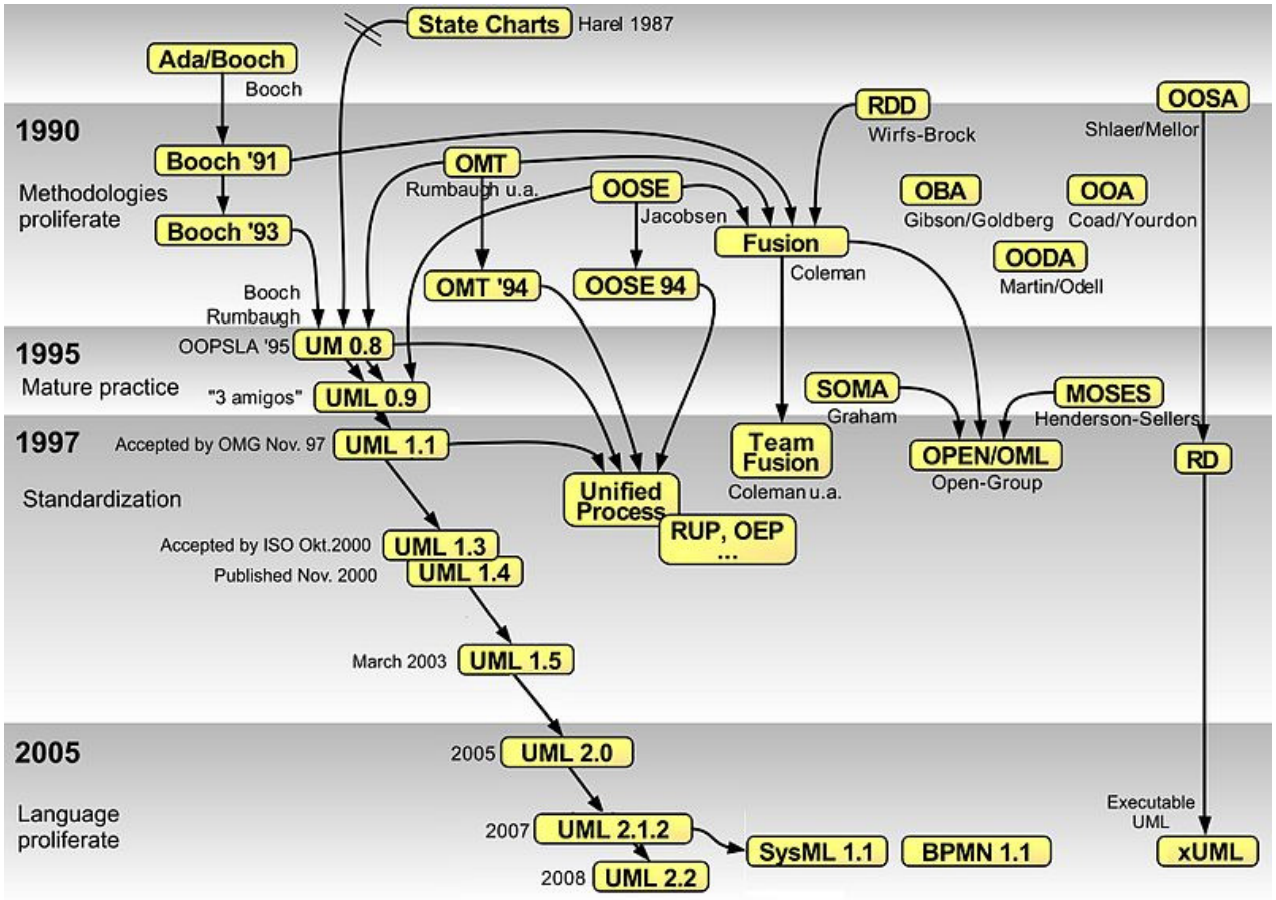
It is a graphical language for visualizing, specifying, constructing and documenting the artifacts of an object-oriented computing system.

Advantages:

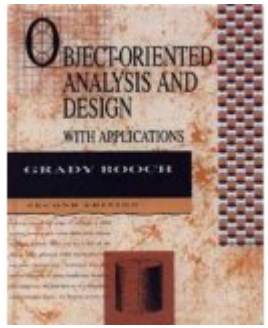
- Fast, efficient and effective development of programs;
- It reveals the desired structure and behavior of the system;
- It allows the visualization and control of the system architecture;
- Better understanding of the system under construction and risk management.

UML

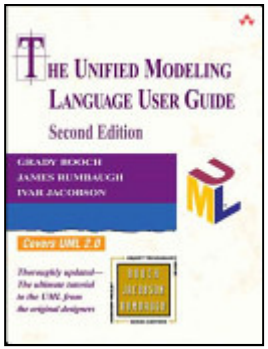
History



1993

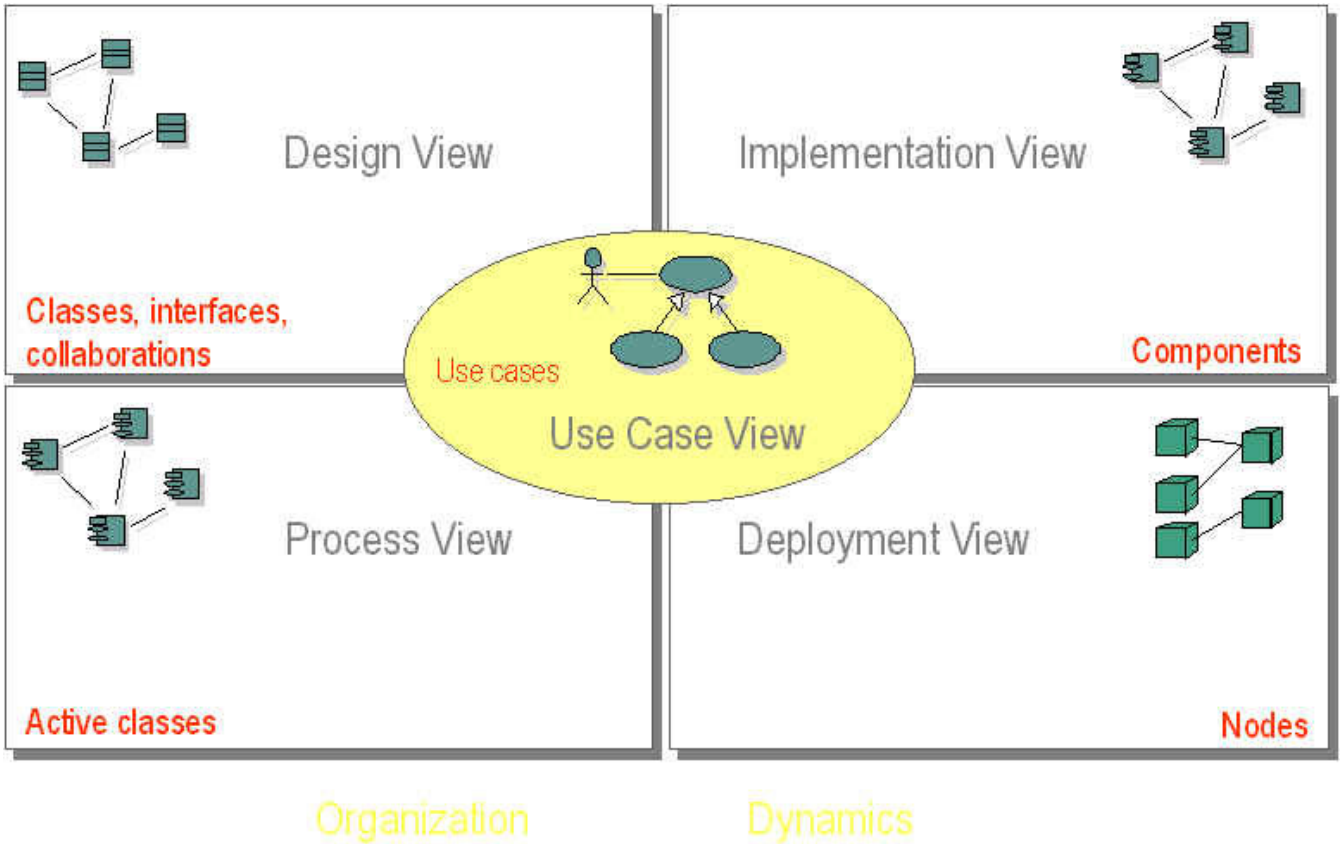


2005

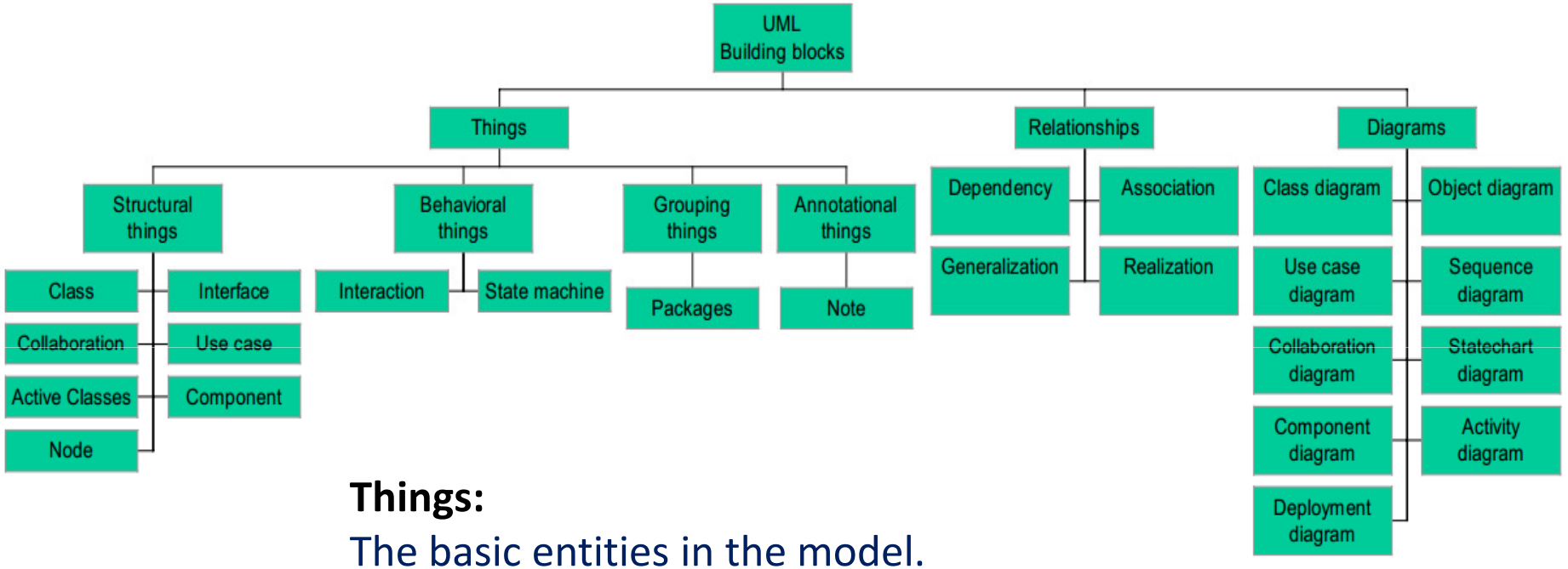


UML

Views (Architecture of an OO System)



UML Building Blocks



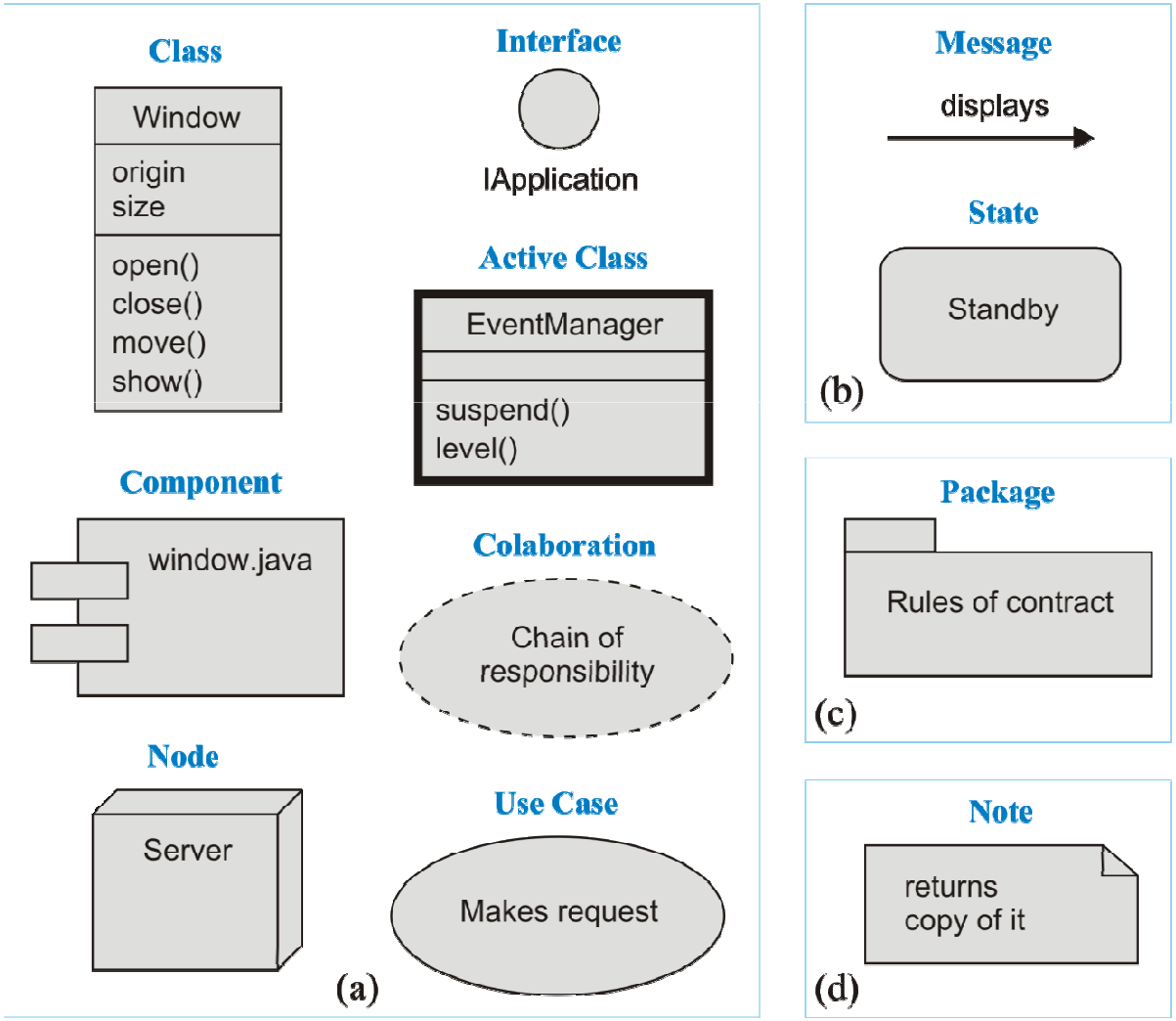
Things:
The basic entities in the model.

Relationships:
Tie things together.

Diagrams:
They are graphs of things and their relationships.

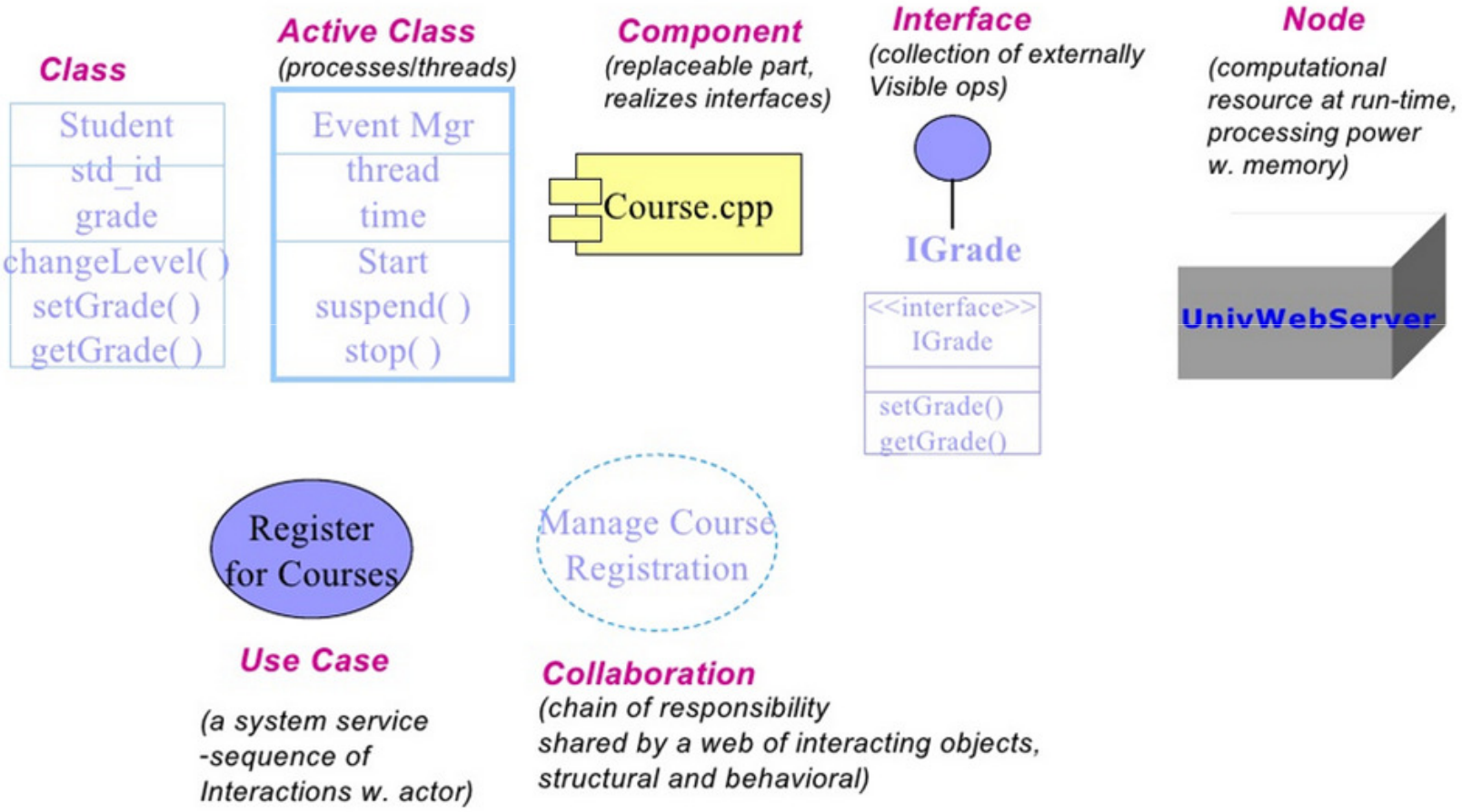
Building Blocks

Things in UML



Building Blocks

Things in UML



Building Blocks

Things: Classes in UML

How to represent this class in UML?

```
#ifndef NUMBER_H
#define NUMBER_H



class Number
{
public:
  Number(int _type);
  ~Number();
  virtual Number sum(Number _n) = 0;
  virtual Number sub(Number _n) = 0;
  virtual Number mul(Number _n) = 0;
  virtual Number div(Number _n) = 0;

protected:
  int m_type;
};

#endif
```

Building Blocks

Things: Classes in UML

Number
m_type
sum(_n) *sub(_n)* *mul(_n)* *div(_n)*

Number
m_type:int
+ sum(_n:Number) : Number *+ sub(_n:Number) : Number* *+ mul(_n:Number) : Number* *+ div(_n:Number) : Number*

```
#ifndef NUMBER_H
#define NUMBER_H



class Number
{
public:
  Number(int _type);
  ~Number();
  virtual Number sum(Number _n) = 0;
  virtual Number sub(Number _n) = 0;
  virtual Number mul(Number _n) = 0;
  virtual Number div(Number _n) = 0;

protected:
  int m_type;
};

#endif
```

Building Blocks

Things: Classes in UML

Number
m_type
sum(_n) *sub(_n)* *mul(_n)* *div(_n)*

Number
m_type:int
+ sum(_n:Number) : Number *+ sub(_n:Number) : Number* *+ mul(_n:Number) : Number* *+ div(_n:Number) : Number*

Real
m_value
sum(_n) sub(_n) mul(_n) div(_n)

Complex
– m_real:double – m_imag:double
+ sum(_n:Number) : Number + sub(_n:Number) : Number + mul(_n:Number) : Number + div(_n:Number) : Number

```
#ifndef NUMBER_H
#define NUMBER_H



class Number
{
public:
  Number(int _type);
  ~Number();
  virtual Number sum(Number _n) = 0;
  virtual Number sub(Number _n) = 0;
  virtual Number mul(Number _n) = 0;
  virtual Number div(Number _n) = 0;

protected:
  int m_type;
};

#endif
```

Building Blocks

Relationship in UML

Association: Set of links between objects.

Aggregation

Composition

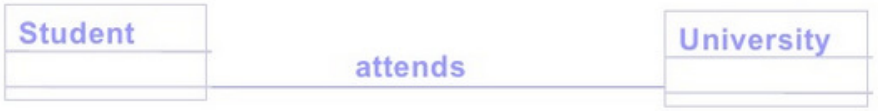
Dependency: Change to one thing will affect the other.

Generalization: Used for inheritance. Encodes "IS-A" relationship.

Realization: A specification of a contract between two entities.

Building Blocks

Relationship in UML



1. Associations

Structural relationship that describes a set of links, a link being a connection between objects. ***variants: aggregation & composition***

2. Generalization

a specialized element (the child) is more specific the generalized element.

3. Realization

one element guarantees to carry out what is expected by the other element.

(e.g, interfaces and classes/components; use cases and collaborations)

4. Dependency

a change to one thing (independent) may affect the semantics of the other thing (dependent).

(direction, label are optional)

Building Blocks

Relationship: between Classes UML

Stack
m_top:int m_elems:*INumber
push(_n:INumber) pop() : INumber isEmpty() : bool show()

INumber
m_type:int
+ sum(_n:INumber) : INumber *+ sub(_n:INumber) : INumber* *+ mul(_n:INumber) : INumber* *+ div(_n:INumber) : INumber*

How are these classes related?

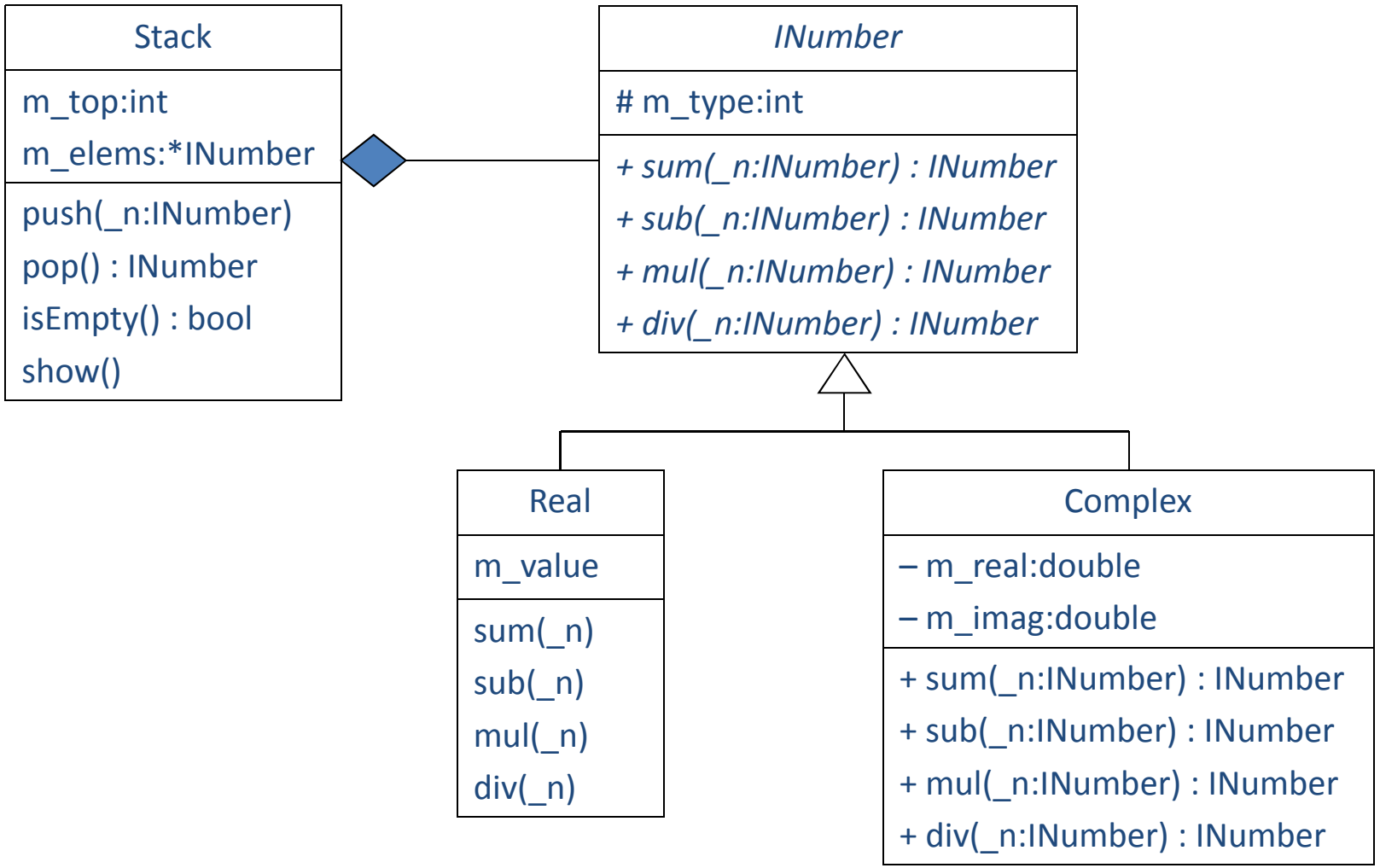
How to represent these relationship in UML?

Real
m_value
sum(_n) sub(_n) mul(_n) div(_n)

Complex
– m_real:double – m_imag:double
+ sum(_n:INumber) : INumber + sub(_n:INumber) : INumber + mul(_n:INumber) : INumber + div(_n:INumber) : INumber

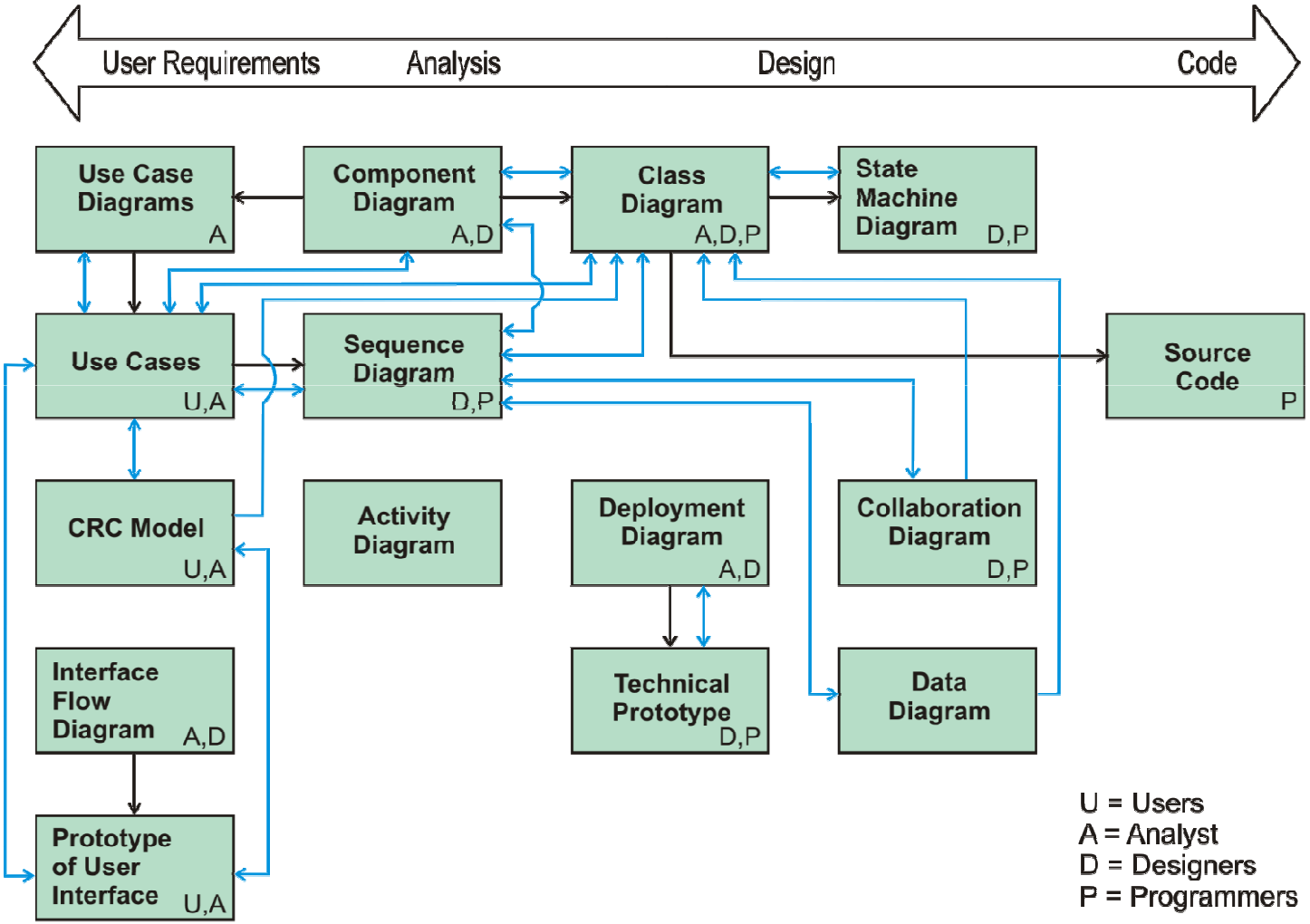
Building Blocks

Relationship: between Classes UML



Building Blocks

Diagrams in UML



CRC: class, responsibility and collaboration

Object Oriented Software Modeling

Object Oriented Modeling

A model is an **abstraction of something** for the purpose of understanding it **before building** it (Rumbaugh, 1994). Because, **real systems** that we want to study are generally very **complex**.

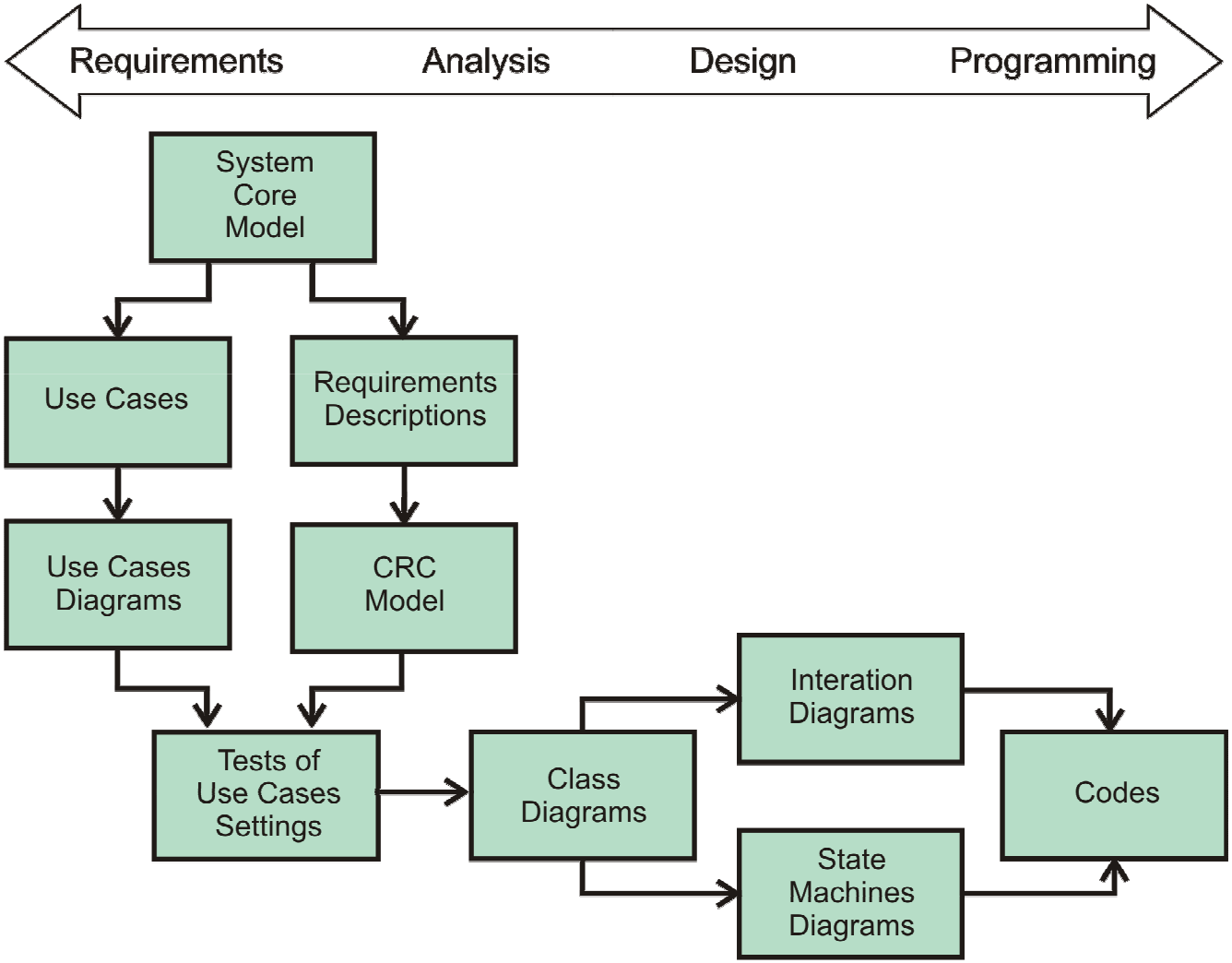
In order to **understand** the real system, we have to **simplify** the system. So a model is an abstraction that **hides non-essential characteristics** of a system and highlights those characteristics, which are pertinent to understand it.

A model can also be understood as a **simplified representation of reality**. A model provides a means for conceptualization and communication of ideas in a **precise and unambiguous form**.

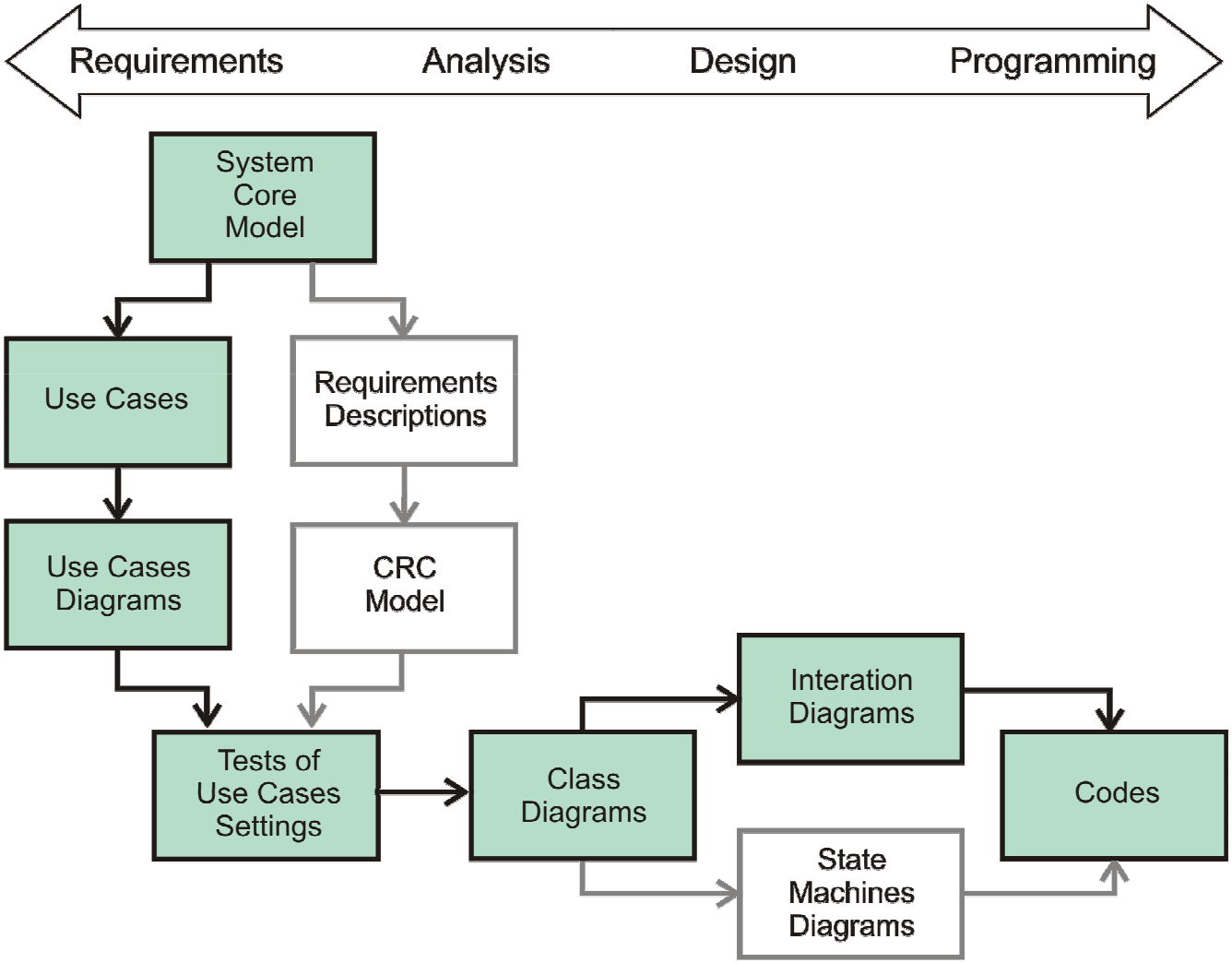
The characteristics of **simplification** and **representation** are **difficult to achieve in the real world**, since they frequently contradict each other. **Thus modeling enables us to deal with the complexity of a system**.

(Source: Nath, 2014 – Lecture Notes on Object-Oriented Methodology)

Object Oriented Modeling



Object Oriented Modeling



Object Oriented Modeling

- **Requirements**
  - List of requirements / needs
  - User Interface
- **Object Oriented Analysis**
  - Use Cases
  - Robustness Diagram
- **Object Oriented Design**
  - Sequence Diagram
  - Class Diagram
- **Object Oriented Programming**

Object Oriented Modeling
of a RPN Calculator

Requirements

It should be possible to insert several numbers on the calculator. The numbers can be integers, real and complex. The real numbers have two decimal places and the complexes also two decimal places in the real and imaginary parts.

It should be possible to perform the four basic operations: addition, subtraction, multiplication and division.

Operations must be carried out with the last two numbers entered in the calculator. Therefore, an operation requires that the users enter with at least two numbers. The result of each operation is a new number which replaces the two numbers used in the operation. The remaining of the numbers is unchanged.

It should be viewed only the last four numbers entered.

Object Oriented Modeling
of a RPN Calculator

User Interface

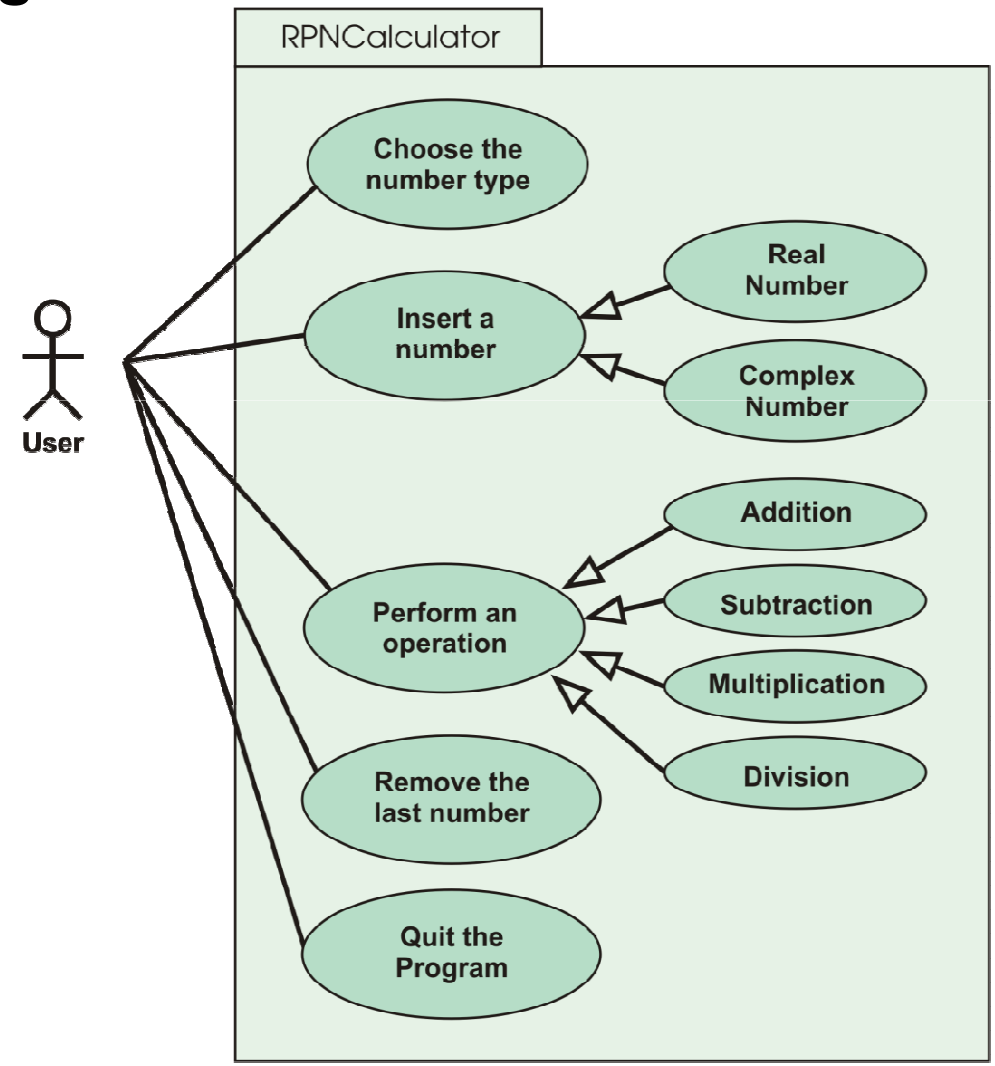
Sketch of the program graphical interface.

There are missing in the sketch the following buttons:

- *enter* ENTER
- delete the last entered number ←
- Switching to different type of number NXT

Object Oriented Analysis
of a RPN Calculator

Use Cases



Object Oriented Analysis
of a RPN Calculator

Use Cases

- **Choose the Number Type**

May be an option held at the beginning of program execution, which will define the behavior of the calculator. During program execution, the user can also press a button to choose the type of number that he wants to work. The numbers that are already in the calculator should be automatically converted to the new format.

- **Enter a number**

The use case "Enter a Number" is initialized when the user presses a button corresponding to the number that he want to enter on the calculator. If the number is of type Integer or Real it just click the button with the number, but if the type is complex it must first insert the real part and then, after a space, the imaginary part.

- **Perform an Operation**

This use case starts when the user presses the corresponding button operation he wants to accomplish. Any operation is performed with the last two numbers entered in the calculator, but the result of course depends on the operation.

- **Remove the last number**

Removes the last number without performing operations. The previous becomes the last.

- **Quit the Program**

This use case starts when the user clicks the close box of the program in the main application window. The values that are in the calculator are lost.

Object Oriented Analysis
of a RPN Calculator

Robustness Diagram

A robustness diagram is basically a simplified UML collaboration diagram.

A initial reading of the use cases suggests that following will be part of the system:

- A single entity object representing the current calculator that the program is working with (`RPN`).
- An arbitrary number of entity objects, each representing one of the numbers (`Number`) that is in the current calculator. This number can still be: integer (`Integer`), real (`Real`) or complex (`Complex`).
- A special data structure to store the numbers, where the last element added to the structure must be the first one to be removed. Therefore, the structure that is most suitable for this application is the stack (`Stack`).
- A boundary object representing the interface between the calculator system and the human user (`RPN_GUI`).
- A controller object that carries out the use cases in response to user gestures on the GUI (`RPNController`). (For a problem of this small size, a single controller is sufficient.)

Object Oriented Analysis
of a RPN Calculator

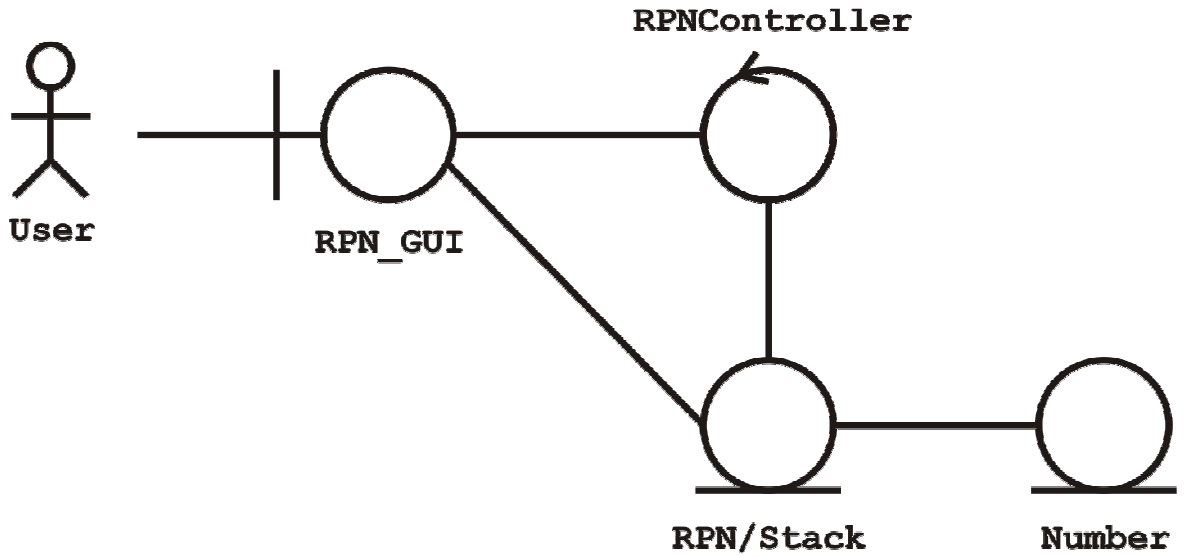
Robustness Diagram

The various use cases work with these objects, as follows:

- *Enter a number* involves taking the new user input, and then tell the RPN object to add a new number with this information in its data structure.

- *Perform an operation* involves removing the last two numbers stored in RPN object, perform the operation with these numbers and display the result on the screen, which is added as a new item in your collection.

- etc…



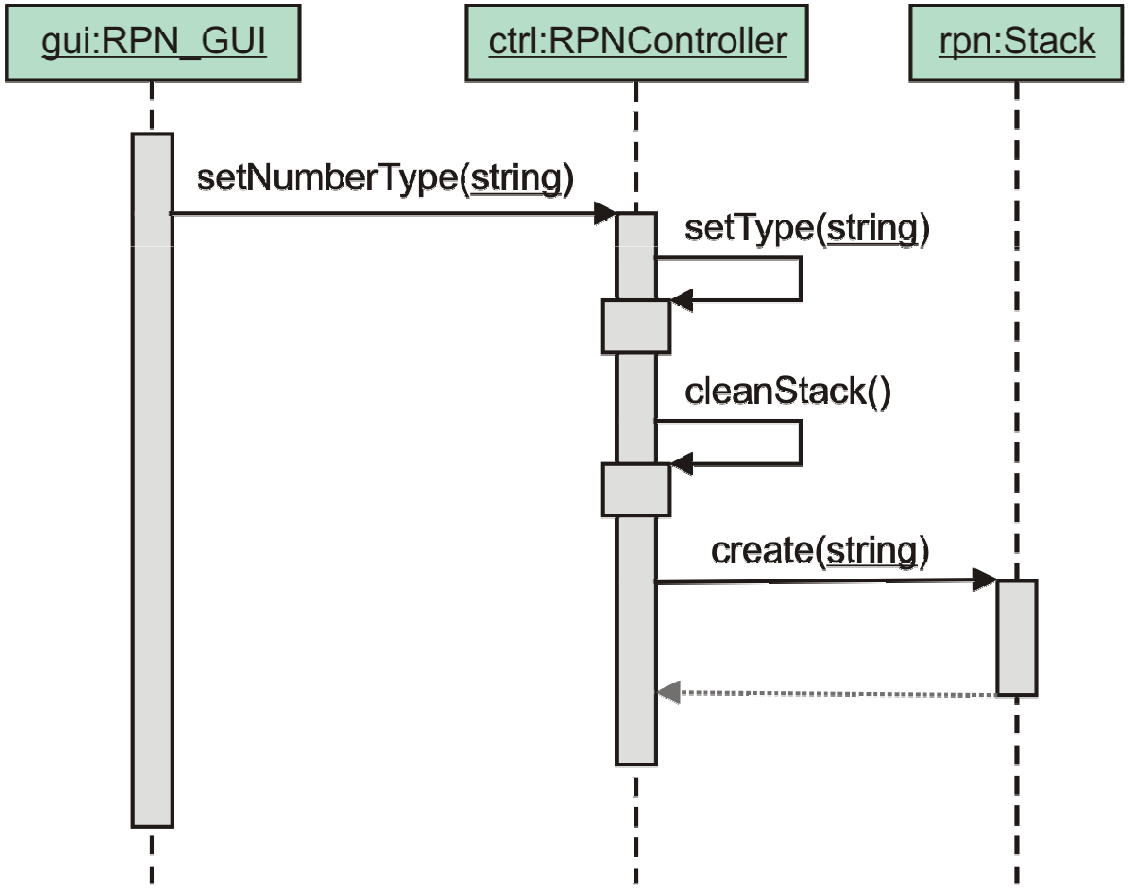
Object Oriented Design
of a RPN Calculator

Sequence Diagram

Each of the use cases discovered in the analysis of the system will be realized by a sequence of operations involving the various objects comprising the system:

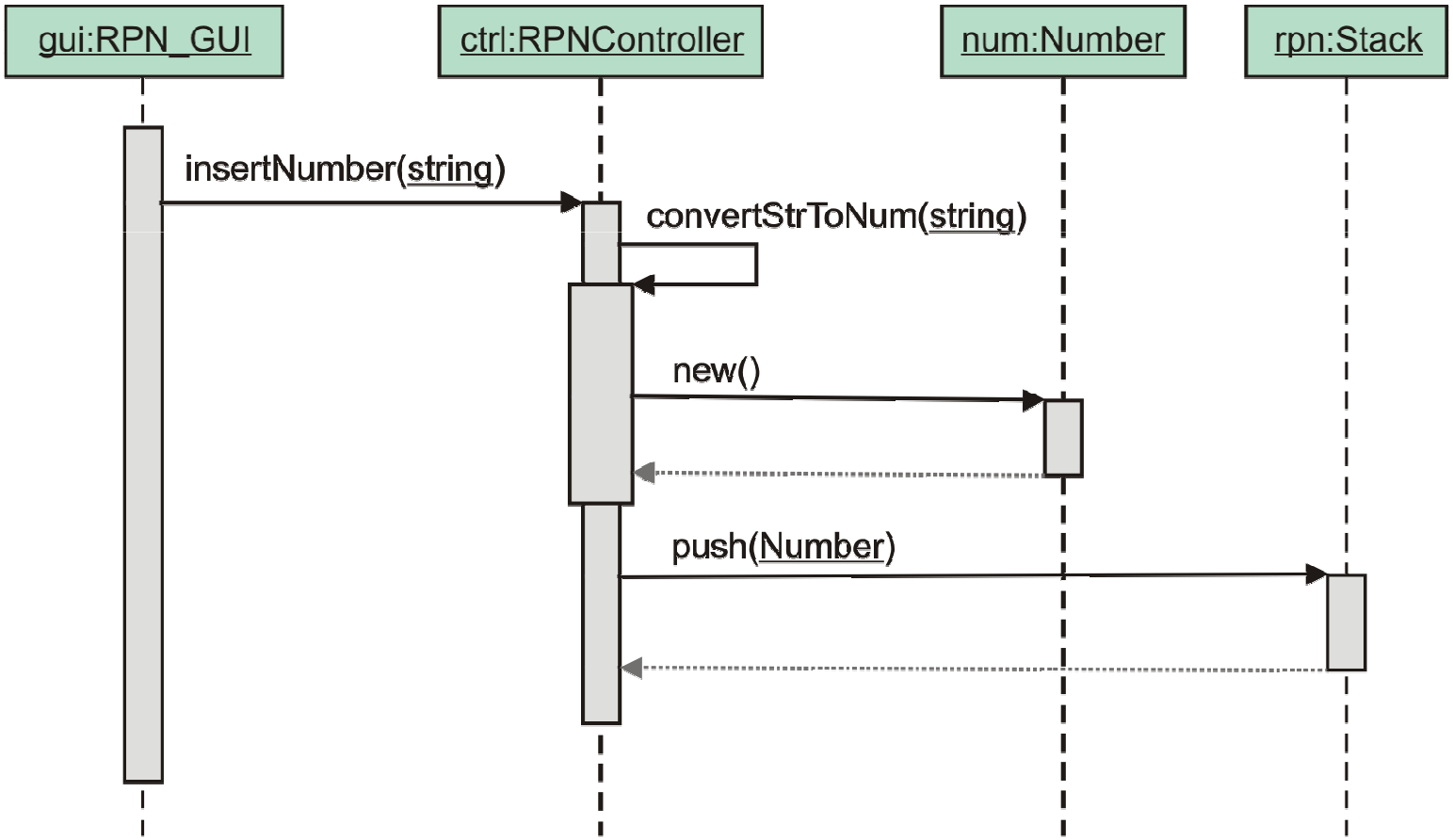
Object Oriented Design
of a RPN Calculator

Sequence Diagram



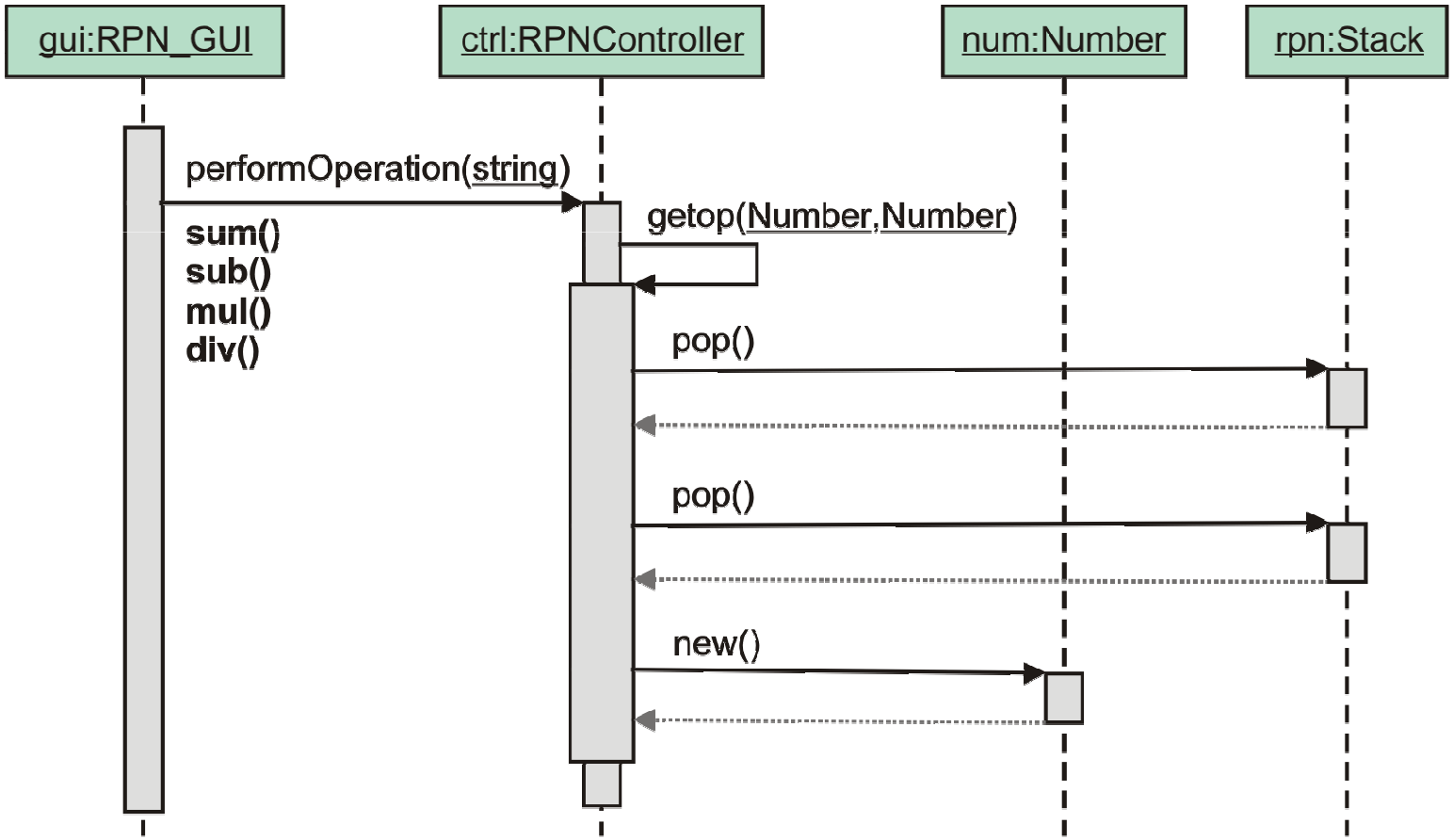
Object Oriented Design
of a RPN Calculator

Sequence Diagram



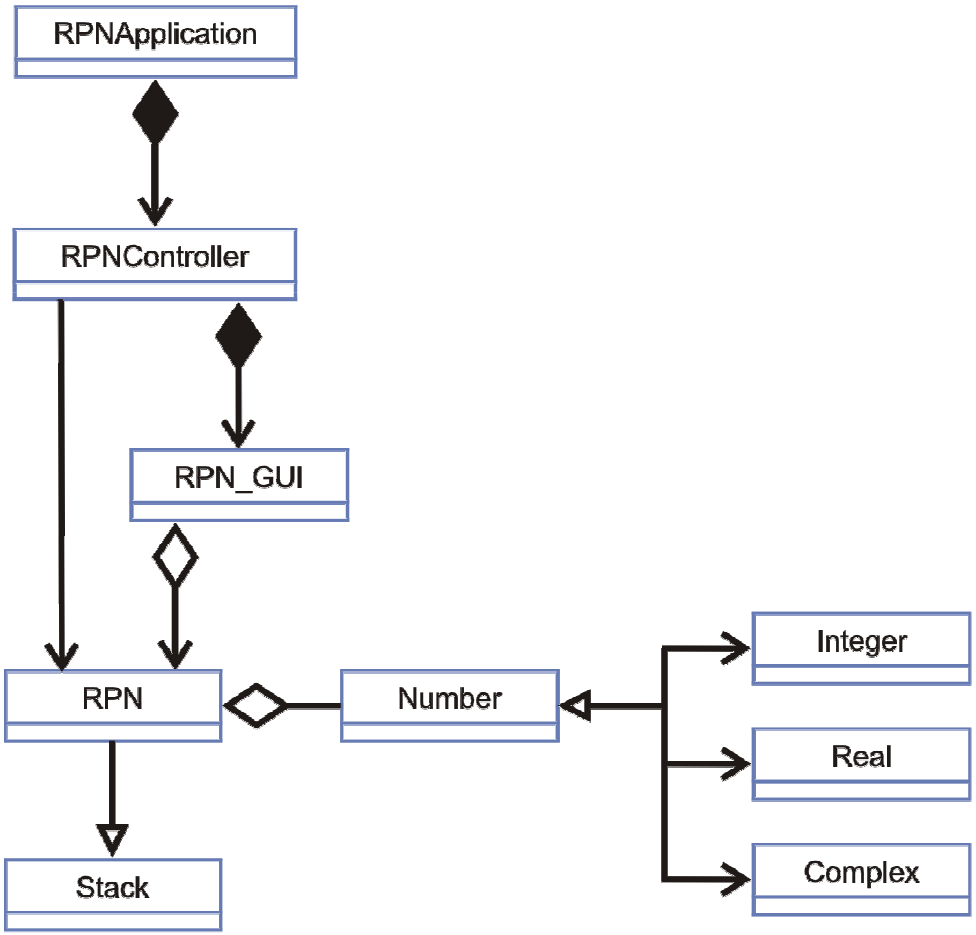
Object Oriented Design
of a RPN Calculator

Sequence Diagram



Object Oriented Design
of a RPN Calculator

Class Diagram



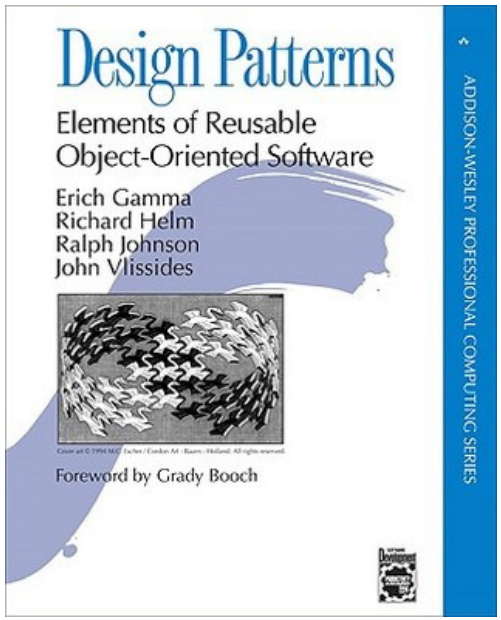
Introduction to Design Patterns

Design Patterns

- Identification of Objects (difficult task)

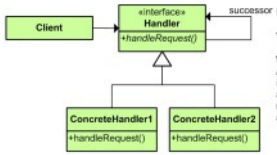
- Techniques for System Decomposition in Objects

- Identification of non Obvious Abstraction



C Abstract Factory
S Adapter
S Bridge
C Builder
B Chain of Responsibility
B Command
S Composite
S Decorator
S Facade
C Factory Method
S Flyweight
B Interpreter
B Iterator
B Mediator
B Memento
C Prototype
S Proxy
B Observer
C Singleton
B State
B Strategy
B Template Method
B Visitor

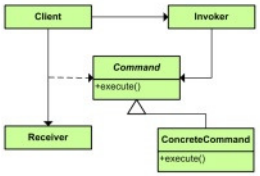
Chain of Responsibility



Type: Behavioral

What it is:
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

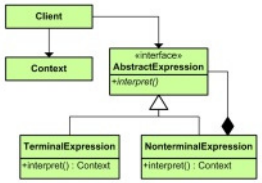
Command



Type: Behavioral

What it is:
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

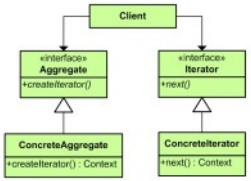
Interpreter



Type: Behavioral

What it is:
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

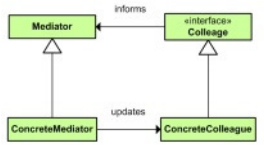
Iterator



Type: Behavioral

What it is:
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Mediator

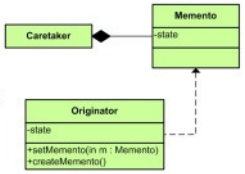


Type: Behavioral

What it is:
Define an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly and it lets you vary their interactions independently.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc..

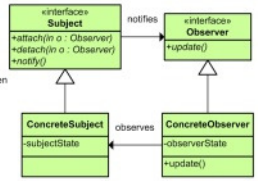
Memento



Type: Behavioral

What it is:
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

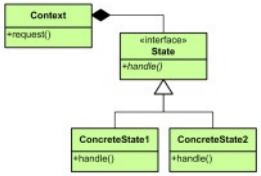
Observer



Type: Behavioral

What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

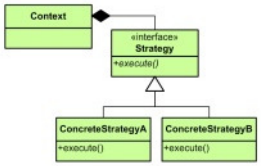
State



Type: Behavioral

What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

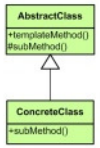
Strategy



Type: Behavioral

What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.

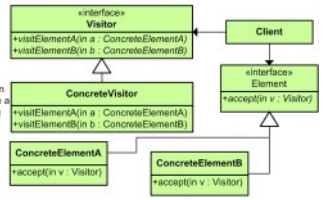
Template Method



Type: Behavioral

What it is:
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

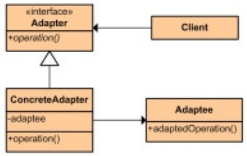
Visitor



Type: Behavioral

What it is:
Represent an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.

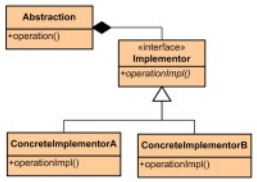
Adapter



Type: Structural

What it is:
Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.

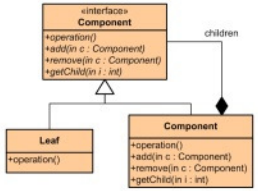
Bridge



Type: Structural

What it is:
Decouple an abstraction from its implementation so that the two can vary independently.

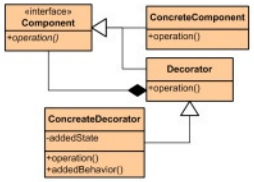
Composite



Type: Structural

What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.

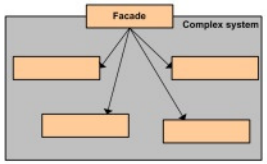
Decorator



Type: Structural

What it is:
Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality.

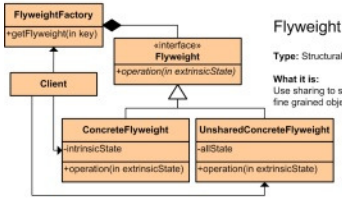
Facade



Type: Structural

What it is:
Provide a unified interface to a set of interfaces in a subsystem. Defines a high-level interface that makes the subsystem easier to use.

Flyweight

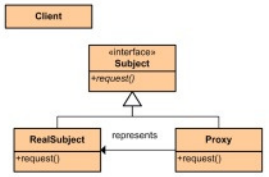


Type: Structural

What it is:
Use sharing to support large numbers of fine grained objects efficiently.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc..

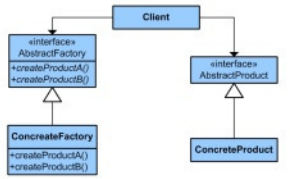
Proxy



Type: Structural

What it is:
Provide a surrogate or placeholder for another object to control access to it.

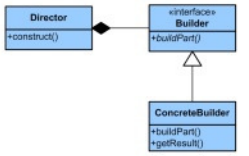
Abstract Factory



Type: Creational

What it is:
Provides an interface for creating families of related or dependent objects without specifying their concrete class.

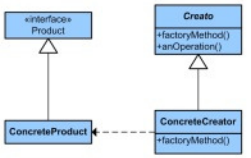
Builder



Type: Creational

What it is:
Separate the construction of a complex object from its representing so that the same construction process can create different representations.

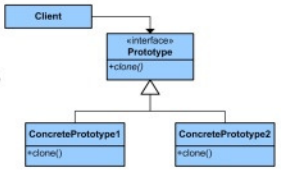
Factory Method



Type: Creational

What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.

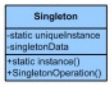
Prototype



Type: Creational

What it is:
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Singleton



Type: Creational

What it is:
Ensure a class only has one instance and provide a global point of access to it.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc..

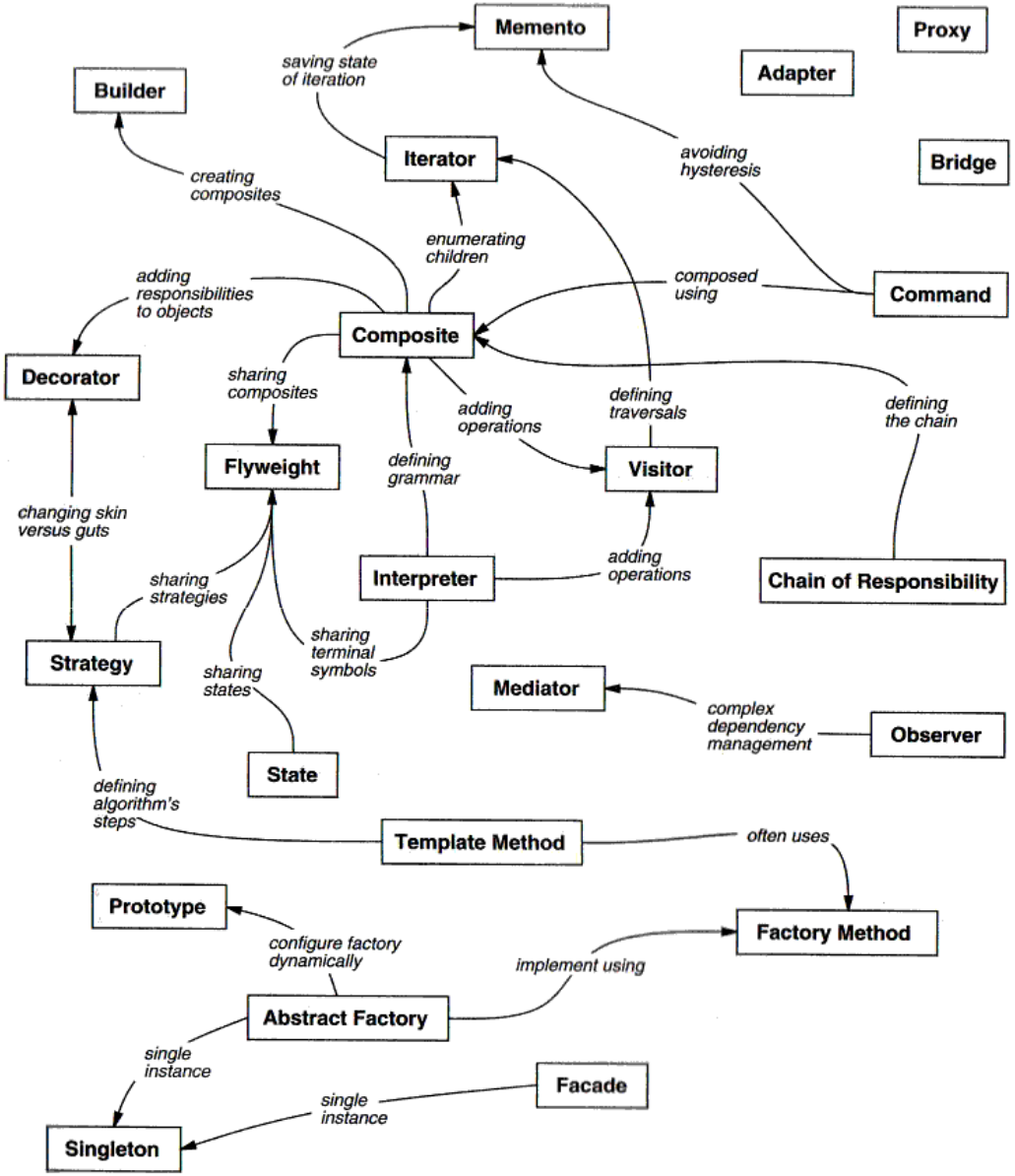


Figure 1.1: Design pattern relationships