Introduction to Software Design

Motivation for the C Object System

Laurent Deniau

CERN – Accelerator Technologies Department

laurent.deniau@cern.ch

March 27, 2008

DRAFT

Content

Outline

Software Development

☞ *Make software that fulfills the requirements*

➠ *How to know the requirements?*

✌ *If you can answer to the question now and forever the task is easy (and you can quit the seminar)*

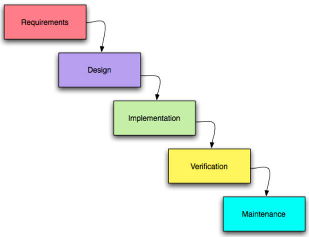
Development Model

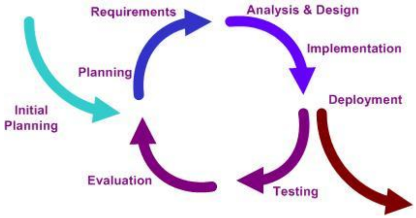
WATERFALL model *(predictive)*

- One large step
- Delivery at the end
- Very expensive refactoring
- Experienced designers
- Emphase documentation
- Risk of broken design

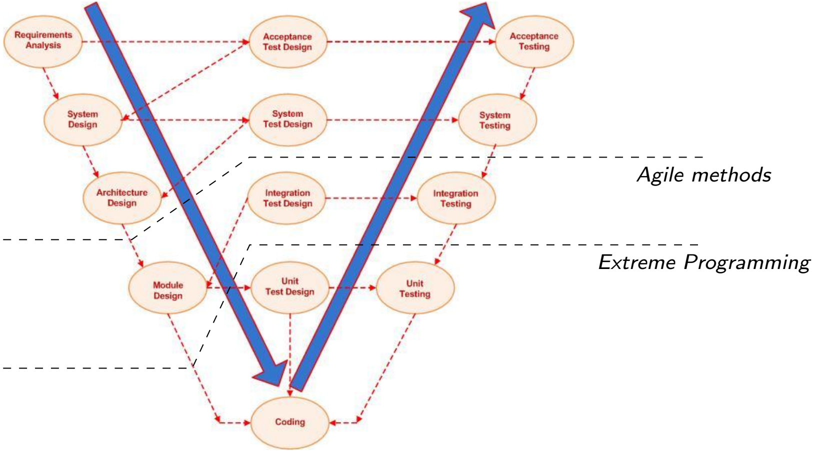
ITERATIVE model *(adaptive)*

- Many short cycles
- Incremental deliveries
- Frequent refactoring
- Experienced developers
- Emphase communication
- Risk of non-convergence

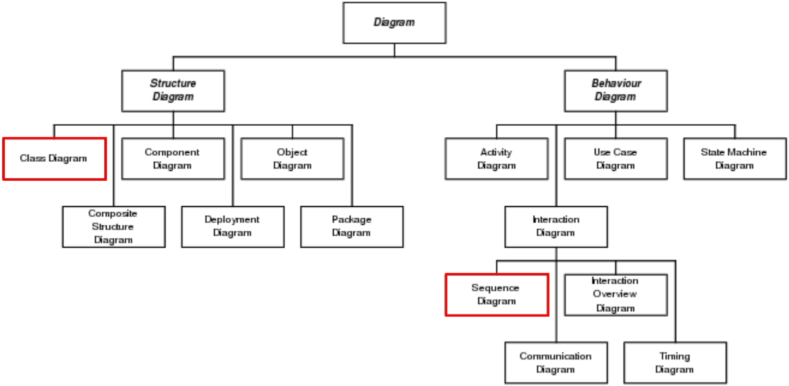




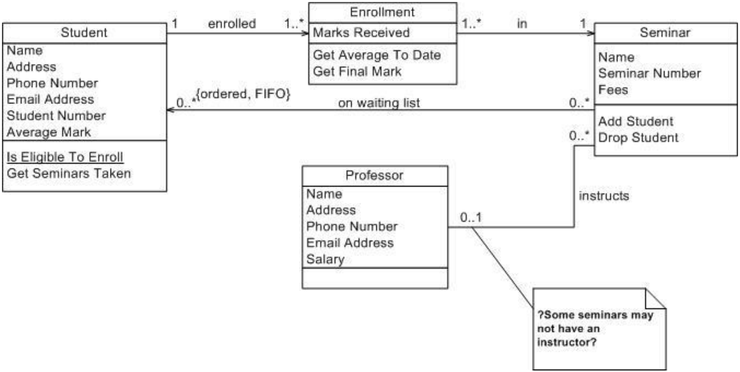
Development and V-Cycle



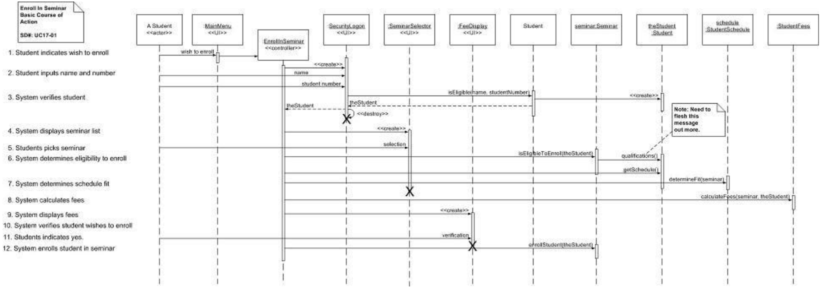
Unified Modeling Language



UML Class Diagram Example



UML Sequence Diagram Example



Software Development Summary

Summary

- Two development models dominate *(with many variants)*
- Predictive models require experienced designers
- Adaptive models require experienced developers
- All models require large team of skilled professionals *(min. 5)*
- Management relies on modeling languages and tools

Outline

Software Design

☞ *Make software that fulfills the requirements*

➠ *How to know the requirements? Don't know!*

☞ *Make software that survives long-term evolution*

➠ *How to design software for evolution? Any recipes?*

Design and Evolution

Design

- Large projects begin with clean and clear design

Evolution

- Design never/cannot anticipate long-term evolution
- Changes unanticipated by design degrade the software quality
- Volatile requirements make the code ugly and unmaintainable
- Complete redesign rarely succeeds because...
  it runs after a moving target

Conclusion

- *Design must anticipate the capacity to evolve*
- *Flexibility and extensibility must be considered from start*

Design and Production

Design

- Design affects the software at all scales and levels
  function, class, module, framework, application, environment

Production

- Developers are rarely expert designers
- Developers take design decisions at nearly each line of code
- Developers should focus on working solutions *(What to do)*
- Developers should not focus on design problems *(How to do)*
- Developers should easily refactor poor design choices

Conclusion

- *Software developers need recipes, patterns and expressiveness*

Poor Design Symptoms *(Saṃsāra)*

RIGIDITY	The software is difficult to change, even in simple ways
FRAGILITY	The software breaks in many places every time it is changed
IMMOBILITY	The software is hard to extend and requires hacks to evolve
REDUNDANCY	The software fails to reuse/be reused by others, leading to duplications
VISCOSITY	The development environment fails to build and test the software efficiently

Good Design Criteria *(Nirvâna)*

SIMPLICITY	The software (code) can be easily understood *The quality of being simple or uncompounded*
FLEXIBILITY	The software can be easily changed *The quality of being adaptable or variable*
EXTENSIBILITY	The software can easily grow *The quality of being extensible or improved*
REUSABILITY	The software can be easily reused and composed *The ability to create new features without modification*
TESTABILITY	The software can be easily tested *The ability to be validated through tests*

Software Design Summary

Summary

- Design cannot anticipate long-term evolution
- Design must provide the capacity to evolve
- Design is mostly about management of dependencies

Outline

Design Principles

OPEN CLOSED Principle

Components should be open for extension
but closed for modification

LEAST KNOWLEDGE Principle

Assume and expose as little as possible about components

DEPENDENCY INVERSION Principle

Depend upon abstractions, do not depend upon concretions

INTERFACE SEGREGATION Principle

Few client-specific interfaces are better than
one general-purpose interface

Keep Cohesion High *(Locality of Knowledge)*

low

- ✘ COINCIDENTAL COHESION
  Components are grouped arbitrarily
- ☹ CATEGORICAL COHESION
  Components are grouped because they do the same kind of things
- ☹ SEQUENTIAL COHESION
  Components are grouped because they operate in the same execution path

- ☺ STRUCTURAL COHESION
  Components are grouped because they operate on the same data
- ✔ FUNCTIONAL COHESION
  Components are grouped because they contribute to a single well-defined task

high

✍ *High cohesion enhances reliability, reusability and understandability*

Keep Coupling Low *(Dependency of Knowledge)*

✘ INTERNAL COUPLING *high*
Components rely on the content of another component

☹ GLOBAL COUPLING
Components share the same global information

☹ EXTERNAL COUPLING
Components share the same external information

☺ SHARED COUPLING
Components share non-overlapping parts of the same information

☺ STRUCTURAL COUPLING
Components share the same information through their arguments

✔ BEHAVIORAL COUPLING
Components share the same messages or events *low*

✍ *Low coupling enhances readability and maintainability*

Design Principles Summary

Summary

- Design Principles are rather abstract
- Cohesion and Coupling are (paired) ordinal metrics of dependencies
- Design Principles say *What to do*, not *How to do*

☞ Developers need recipies about *How to do*

Outline

Definitions

Software Components

Reusable black boxes *(function, class, module)*

Design Patterns

Reusable design templates to manage components dependencies

Architectural Patterns

Large scale Design Patterns

Design Patterns

- ☺ Design Patterns describe how to solve design problems
- ☺ Design Patterns are workarounds for missing language features
  ex. meta-class, closure, multi-method, delegation, reflection, contract
- ☺ Design Patterns are based on widely available paradigms
  ex. aggregation, interface, inheritance, polymorphism
- ☺ Design Patterns follow the Design Principles

- ☹ Design Patterns increase flexibility by adding levels of indirections
- ☹ Design Patterns do not build reusable components
- ☹ Design Patterns increase complexity and redundancy

Object-Oriented Design Patterns

Creational Patterns

Abstract/decouple/organize objects creation

ex. Abstract Factory, Builder, Factory Method, Prototype, Singleton

Structural Patterns

Abstract/decouple/organize objects composition and connection

ex. Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy

Behavioral Patterns

Abstract/decouple/organize objects collaboration and communication

ex. Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template, Visitor

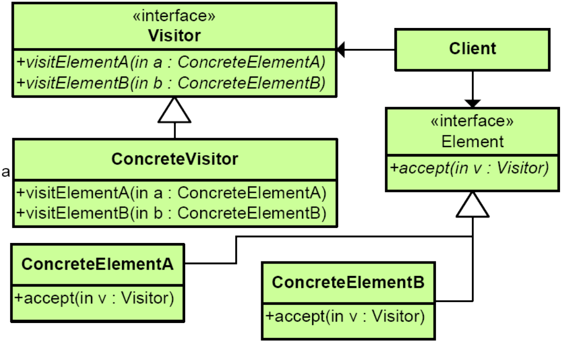
and many more...

Design Pattern Example

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.



Improving Pattern-Oriented Design

- COMPONENTIZATION
  Turning patterns into components to increase effective reusability
- CONCEPTUALIZATION
  Making patterns more abstract/generic to improve the design
- ANTI-PATTERNS *(100+)*
  Patterns to refactor poor/intensive usage of Design Patterns

Development Cycle and Pattern-Oriented Design

1. Analyse and design the components required by the software
2. Select/adapt the Design Patterns to manage the components
3. Implement the design
4. Test the implementation
5. Iterate the process until the requirements are fulfilled

☞ *Pattern-Oriented Design relies on large iterations*

Pattern-Oriented Design Summary

Summary

- Design Patterns explain *How to do*
- Design Patterns relax components dependencies
- Design Patterns increase software complexity

✍ Be aware of the complexity and limitations of Design Patterns

Outline

Definitions

Software Components

Reusable black boxes *(function, class, module)*

Programming Concepts

Expressive programming paradigms to build generic components

Design with Concepts

Simple recipies and metrics to drive the design implicitly

Programming Concept – Type/Object

- **Type**
  Set of values, properties and operations
- **Meta Type**
  Type which defines a type
- **Abstract Type**
  Type with undefined values or operations (incomplete type)
- **Composite Type**
  Type which combines/groups other types (structure)
- **Derived Type**
  Type which includes/extends other types (base types)

polymorphism/specialization

- **Parametric Type**
  Type with definitions parametrized by types
- **Polymorphic Type**
  Type bound to several specializations (derived types)

Programming Concept – Function/Behavior

- **Function**
  Set of statements, expressions and arguments
- **Function Composition** $f(x) = (g \cdot h)(x)$
  Ability to compose functions
- **Function Closure** $f(x) = ax + b\big|_{a,b}$
  Ability to bind functions to their free variables

polymorphism/specialization

- **Parametric Function**
  Function with definitions parametrized by arguments type
- **Polymorphic Function**
  Function bound to several specializations (by arguments type)

Programming Concept – Dependency

- **Encapsulation**
  Ability to reduce dependencies by hiding/protecting information
  - Expose *(stable)* interface and abstract types
  - Hide *(unstable)* implementation and concrete types

dynamic relationship

- **Indirection**
  Ability to reach a target through a key/reference/pointer (wide sens)
- **Aggregation**
  Ability to group objects into collections
- **Delegation**
  Ability to defer/redirect/forward function dispatch
- **Reflection**
  Ability to reify meta-information
  - *Introspection*, ability to read/execute meta-data (interpreter)
  - *Intercession*, ability to modify meta-data (run-time optimization)

Development Cycle and Concept-Oriented Design

1. Analyse and design one component required by the software
2. Implement the component using the right concepts to make
   - it reusable and composable *(small orthogonal services)*
   - its implementation flexible *(closed)*
   - its interface extensible *(open)*
   - it easy to test *(unit tests)* and use *(use cases)*
3. Iterate the process until the requirements are fulfilled

☞ *Concept-Oriented Design relies on small iterations (Agile methods)*

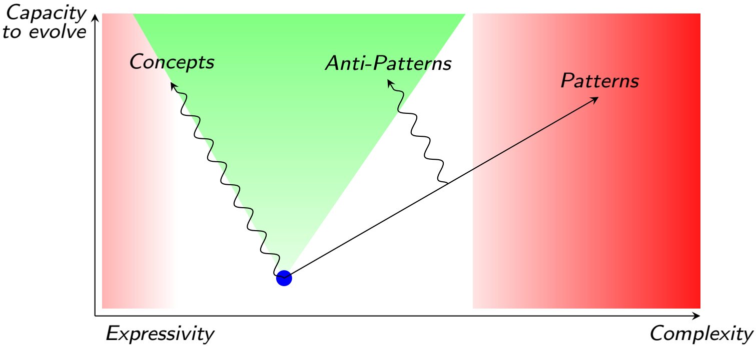
Concept-Oriented Design Summary

Summary

- Concepts do not explain *How to do (require experience)*
- Concepts are more generic than Patterns *(lower level)*
- Concepts solve components dependencies *(require experience)*
- Concepts provide components reusability *(more orthogonal)*
- Concepts decrease the software complexity *(less indirections)*

✍ Be aware of the concepts supported by your programming language

Concept-Oriented vs Pattern-Oriented Design



- ☞ *P-O Design relies on expressive paradigms when "code smells"*
- ☞ *C-O Design relies on Design Patterns when implicit design fails*

Outline

Design and Programming Languages

STRUCTURED Programming

Enforce structured code

ex. C, Fortran, Pascal, Perl

OBJECT-ORIENTED Programming

Enforce type composition, derivation and mutability (states)

ex. Ada, C++, C#, D, Eiffel, Java, Objective-C, Python, Scala, Smalltalk

FUNCTIONAL Programming

Enforce functional composition, closure and immutability (values)

ex. Erlang, F#, Haskell, Lua, OCaml, Scala, Scheme, SML

LOGIC Programming

Enforce relations and constraints

ex. Prolog, SQL

Object-Oriented Programming Languages Terminology

- CLASS
  Meta-type defining an object type
- META CLASS
  Meta-type defining a class type
- ABSTRACT CLASS/INTERFACE
  Class defining an incomplete polymorphic type/set of operations
- SINGLE/MULTIPLE INHERITANCE
  Derived class extending one/many base class(es)
- METHOD/MULTI-METHOD
  Function specialized for one/many polymorphic argument(s)
- ITERATOR/FUNCTOR
  Pointer-like/Function-like object

Typed Programming Languages Terminology

- **Strong typing**
  Types cannot be implicitly converted (no coercion)
- **Static typing**
  Types are known at compile-time
- **Dynamic typing**
  Types are known at run-time
- **Duck typing**
  Types are characterized by their behavior
  - ✘ *"If it walks and quacks like a duck, it is a duck"*
  - ✔ *"If it walks and quacks like a duck, it behaves like a duck"*
- **Subtyping**
  Projection of derived types to base types
- **Type punning**
  Prog. technique that subverts or circumvents the type system

Programming Languages Expressivity

Language	Statements ratio	Lines ratio
C	1	1
C++	2.5	1
Fortran	2.5	0.8
Java†	2.5	1.5
Perl†	6	6
Smalltalk†	6	6.25
Python†	6	6.5

† *Interpreted languages*

Languages with dynamic duck typing

Programming Languages Summary

Summary

- OOP languages favor states and extensibility
- FP languages favor values and reusability
- Recent languages try to unify OO and Functional paradigms
- Languages supporting dynamic duck typing are more expressive

Outline

Subtyping

explicit subtyping in C/C++

```
struct Vect2D { // base type
  double x, y;
};

struct Vect3D {
  Vect2D xy;    // type composition
  double z;
};

double norm2(Vect2D* v) {
  return v->x * v->x + v->y * v->y;
}

int main(void) {
  Vect3D v;
  v.xy.x = 1, v.xy.y = 2, v.z = 3;
  norm2(           &v.xy ); // intrusive
  norm2( (Vect2D*) &v    ); // unsafe
}
```

implicit subtyping in C++

```
struct Vect2D { // base type
  double x, y;
};

struct Vect3D : // derived type
  Vect2D {
  double z;
};

double norm2(Vect2D* v) {
  return v->x * v->x + v->y * v->y;
}

int main() {
  Vect3D v;
  v.x = 1, v.y = 2, v.z = 3;

  norm2( &v ); // v is projected to Vect2D
}
```

- ☞ *Subtyping is a projection of derived type to base type (up cast)*
- ☞ *Up cast reduces static visibility of members (data or functions)*

Subtyping + Polymorphism

subtyping + monomorphism in C++

```
struct Vect2D {  // base type
  double x, y;
  double norm2() {
    return x*x + y*y;
  }

};

struct Vect3D : Vect2D {  // derived type
  double z;
  double norm2() {
    return x*x + y*y + z*z;
  }
};

int main() {
  Vect3D v;  v.x = 1, v.y = 2, v.z = 3;
  Vect2D& vp = v; // v is projected to Vect2D
  vp.norm2();      // return 5 (z is lost)
}
```

subtyping + polymorphism in C++

```
struct Vect2D {  // polymorphic base type
  double x, y;
  virtual double norm2() {
    return x*x + y*y;
  }
  virtual ~Vect2D() { } // avoid memory leaks
};

struct Vect3D : Vect2D {  // derived type
  double z;
  virtual double norm2() {  // override
    return x*x + y*y + z*z;
  }
};

int main() {
  Vect3D v;  v.x = 1, v.y = 2, v.z = 3;
  Vect2D& vp = v; // v is projected to Vect2D
  vp.norm2();      // return 14 (back to Vect3D)
}
```

- ☞ *Polymorphic types cancel the projection by overriding base methods*
- ☞ *Polymorphism must be anticipated in the design of base types*

Static Duck Typing

static duck typing in C++

```
template <typename T>
double norm2(T& a) {     // parametric function (virtual is forbidden)
  return a.norm2();
}

int main() {
  Vect3D v;  v.x = 1, v.y = 2, v.z = 3;
  norm2(v);              // return 14 (Vect3D monomorphic instantiation)
  Vect2D& vp = v;        // v is projected to Vect2D
  norm2(vp);             // return 5 or 14 (Vect2D monomorphic instantiation)
  norm2(complex(3.0)); // compile-time error, undefined member function std::complex<double>.norm2()
}
```

- ☞ *Static typing errors are detected at compile-time*
- ☞ *Static duck typing errors are detected at compile-time instantiation*
- ☞ *Parametric functions are not polymorphic functions* *(hard to implement)*

✍ *Dynamic and static features do not always play well together*

Dynamic Duck Typing

dynamic duck typing in Objective-C

```
@class String, Number; // weak coupling

int main(void) {
  id obj[2]; // array of 2 objects

  // hereafter, undefined messages should raise a run-time error

  obj[0] = [String newWithStr: ''hello world''];
  obj[1] = [Number newWithDbl: 10.0];

  [obj[0] print];    // C++ equivalent: obj[0]->print()
  [obj[1] print];

  [obj[0] norm2];    // should raise a run-time error

  [obj[0] release]; // C++ equivalent: delete obj[0];
  [obj[1] release];
}
```

- ☞ *Dynamic duck typing errors are detected at run-time*
- ☞ *Method dispatch does not require any static visibility with DDT*

Static Typing and Dynamic Typing

static vs dynamic typing in C++

```
struct A {
  virtual void f() = 0;
          void g() { f(); }
  virtual     ~A() { g(); }  // "Effective C++", item 9
};

struct B : A {
  virtual void f() { }  // override
};

int main() {
  B b;
} // "pure virtual method called" when b is destroyed
```

- *Static typing does not ensure correctness*
  - *no type system is bulletproof, the code must be tested!*

discussed later

- Liskov Subtitution Principle (subclassing vs. subtyping)
- Covariant and contravariant types

Types and Polymorphism Summary

Summary

- Subtyping is a projection except for polymorphic types
- Static typing allow to detect type errors at compile-time but
  - it can reject valid programs
  - it can accept invalid programs
- Duck typing allow to write simpler and more expressive code but
  - static type errors are detected at compile-time instantiation
  - dynamic type errors are detected at run-time
- Software must be tested

Outline

Data Type

from matrix.h

```
typedef struct Matrix Matrix;

struct Matrix {
  size_t nrow, ncol;
  double val[];
};

Matrix* mtx_new(size_t nrow, size_t ncol, double val);          // constructor
void    mtx_del(Matrix* mtx);                                   // destructor
size_t  mtx_row(Matrix* mtx);
size_t  mtx_col(Matrix* mtx);
double  mtx_get(Matrix* mtx, size_t row, size_t col);
void    mtx_set(Matrix* mtx, size_t row, size_t col, double val);
void    mtx_add(Matrix* mtx, Matrix* mtx2);                     // mtx += mtx2
```

☞ *simplicity 5/5, flexibility 1/5, extensibility 1/5*

Data Type *(cont.)*

from matrix.c

```
Matrix* mtx_new(size_t nrow, size_t ncol, double val) {
  Matrix* mtx = malloc(sizeof *mtx + nrow * ncol * sizeof *mtx->val); assert(mtx);

  mtx->nrow = nrow;
  mtx->ncol = ncol;
  for (size_t i=0; i < nrow * ncol; i++)
    mtx->val[i] = val;
  return mtx;
}

double mtx_get(Matrix* mtx, size_t row, size_t col) {
  assert( row < mtx->nrow && col < mtx->ncol );

  return mtx->val[row * mtx->ncol + col];
}

void mtx_add(Matrix* mtx, Matrix* mtx2) {
  assert( mtx->nrow == mtx2->nrow && mtx->ncol == mtx2->ncol );

  for (size_t i=0; i < mtx->nrow * mtx->ncol; i++)
    mtx->val[i] += mtx2->val[i];
}
```

Abstract Data Type

from matrix.h

```
typedef struct Matrix Matrix; // incomplete type

typedef enum {
  MTX_DENSE, MTX_DIAG, MTX_UTRIANG, MTX_LTRIANG, MTX_BANDED, MTX_SPARSE
} MTX_TYPE;

Matrix* mtx_new(MTX_TYPE type, size_t nrow, size_t ncol, double val);
void    mtx_del(Matrix* mtx);
size_t  mtx_row(Matrix* mtx);
size_t  mtx_col(Matrix* mtx);
double  mtx_get(Matrix* mtx, size_t row, size_t col);
void    mtx_set(Matrix* mtx, size_t row, size_t col, double val);
void    mtx_add(Matrix* mtx, Matrix* mtx2);
```

☞ *simplicity 3/5, flexibility 3/5, extensibility 2/5*

Abstract Data Type *(cont.)*

from matrix.c

```
struct Matrix {  // private definition
  MTX_TYPE type; // state to emulate polymorphism
  size_t nrow, ncol;
  double val[];
};

Matrix* mtx_new(MTX_TYPE type, size_t nrow, size_t ncol, double val) {
  switch(type) {
  case MTX_DENSE: // allocate, initialize and return a dense matrix
  case MTX_DIAG : assert(nrow == ncol); // ...a diagonal matrix
  // etc...
  }
}

void mtx_add(Matrix* mtx, Matrix* mtx2) {
  switch(mtx->type) {
  case MTX_DENSE:
    switch(mtx2->type) {
  // etc... for a total of 6x6 cases of adding mtx2 to mtx!
  }
}
```

Abstract Data Type + Polymorphism

from matrix.h

```
typedef struct Matrix Matrix;

struct Matrix {
  struct Matrix_Interface *i;  // pointer to services
  struct Matrix_Data      *d;  // pointer to data (ADT)
};

struct Matrix_Interface {      // table of services
  void   (*del)(Matrix* mtx);  // polymorphic destructor
  size_t (*row)(Matrix* mtx);
  size_t (*col)(Matrix* mtx);
  double (*get)(Matrix* mtx, size_t row, size_t col);
  void   (*set)(Matrix* mtx, size_t row, size_t col, double val);
  void   (*add)(Matrix* mtx, Matrix* mtx2);
};

// inlined wrappers
static inline void   mtx_del(Matrix* mtx)               { mtx->i->del(mtx); }
static inline size_t mtx_row(Matrix* mtx)        { return mtx->i->row(mtx); }
// etc...
static inline void   mtx_add(Matrix* mtx, Matrix* mtx2) { mtx->i->add(mtx,mtx2); }
```

☞ *simplicity 2/5, flexibility 4/5, extensibility 3/5*

Abstract Data Type + Polymorphism *(cont.)*

from matrix-dense.h

```
#include "matrix.h"

// constructor(s)
extern Matrix* DenseMatrix(size_t nrow, size_t ncol, double val);

// table of services (and type identifier)
extern struct Matrix_Interface DenseMatrixInterface;
```

from matrix-dense-d.h

```
#include "matrix-dense.h"

#ifdef MATRIX_DENSE_C
struct Matrix_Data      { // private data type (for matrix-dense.c only)
#else
struct DenseMatrix_Data { // semi-private data type (shared between matrix implementations)
#endif
  size_t nrow, ncol;
  double val[];
};
```

Abstract Data Type + Polymorphism *(cont.)* 3/5

from matrix-dense.c

```
#define MATRIX_DENSE_C
#include "matrix-dense-d.h"

Matrix* DenseMatrix(size_t nrow, size_t ncol, double val) {
  Matrix* mtx = malloc(sizeof *mtx); assert(mtx);
  mtx->i = &DenseMatrixInterface; // set matrix services (= type identifier)
  mtx->d = malloc(sizeof *mtx->d + nrow*ncol*sizeof *mtx->d->val); assert(mtx->d);
  // initialize data of mtx->d
  return mtx;
}

static void add(Matrix* mtx, Matrix* mtx2) { // private implementation
  if (mtx2->i == &DenseMatrixInterface)
    // add dense matrix mtx2 to dense matrix mtx
  else if (mtx2->i == &DiagonalMatrixInterface)
    // add diagonal matrix mtx2 to dense matrix mtx
  // etc... for a total of 6 cases of adding mtx2 to mtx
}

// table of services (and type identifier)
struct Matrix_Interface DenseMatrixInterface = {
  del, row, col, get, set, add
};
```

Programming Techniques Summary

Summary

- Interfaces are an artefact of static typing ☹
- Interfaces provide a coherent abstraction of services ☺
- Interfaces provide flexibility by decoupling the interface from the implementation (encapsulation) ☺
- Interfaces provide extensibility by allowing new types to implement the services (polymorphism) ☺

Outline

Interface

from matrix.hpp

```
struct Matrix {
  virtual       ~Matrix() = 0; // polymorphic destructor
  virtual size_t row() = 0;
  virtual size_t col() = 0;
  virtual double get(size_t row, size_t col) = 0;
  virtual void   set(size_t row, size_t col, double val) = 0;
  virtual void   add(Matrix& mtx2) = 0; // usage: mtx.add(mtx2);
};
```

from matrix-dense.hpp

```
class DenseMatrix : public Matrix {
public:
  DenseMatrix(size_t nrow, size_t ncol, double val); // constructor
  // ...
private: // weak encapsulation
    size_t nrow_, ncol_;
    std::valarray<double> val_;
};
```

☞ *simplicity 3/5, flexibility 3/5, extensibility 3/5*

Interface *(cont.)*

from matrix-dense.cpp

```
DenseMatrix::DenseMatrix(size_t nrow, size_t ncol, double val)
  : nrow_(nrow), ncol_(ncol), val_(val, nrow*ncol) { // initializer list
}

DenseMatrix::~Matrix() { // trivial
}

double DenseMatrix::get(size_t row, size_t col) {
  assert( row < this->nrow_ && col < this->ncol_ ); // explicit use of this

  return val_[row*ncol_ + col];                     // implicit use of this
}

// other services...

void DenseMatrix::add(Matrix& mtx2) {
  if ( dynamic_cast<DenseMatrix*>(&mtx2) )          // downcast, static type was lost
    // add dense matrix mtx2 to dense matrix mtx (this)
  else if ( dynamic_cast<DiagonalMatrix*>(&mtx2) )  // downcast, static type was lost
    // add diagonal matrix mtx2 to dense matrix mtx (this)
  // etc... for a total of 6 cases of adding mtx2 to mtx
}
```

Liskov Substitution Principle

Liskov Substitution Principle

Subclasses should be substitutable for their base classes

Static Types and Polymorphism

- **Covariance** downcast of types are allowed in specialization
- **Contravariance** up cast of types are allowed in specialization
- **Subclassing** Subtyping + Polymorphism ⇒ can break the LSP

- ☞ *Covariance enlarges static visibility (downcast)*
- ☞ *Parametric types must manage variance dependencies (annotations in Scala)*
- ☞ *Liskov Substitution Principle is difficult to apply in practice*

☺ *Method selector(s) (`this`) are always covariant (polymorphism)*

Application of Design Principles *(abstractions)*


```
struct Comparable {
  virtual int compare(Comparable&) = 0;
  virtual ~Comparable() { }
};
struct Copiable {
  virtual void copy(Copiable&) = 0;
  virtual ~Copiable() { }
};
struct Clonable : virtual Copiable {  // virtual inheritance avoid (futur) duplication
  virtual Clonable* clone() = 0;
  virtual ~Clonable() { }
};
struct Swapable : virtual Clonable {
  virtual void swap(Swapable& s) {      // generic implementation
    Clonable *c = s.clone();            //  s            must be Clonable
    s.copy(*this), this->copy(*c);      //  s, this and c must be Copiable
    delete c;                           // ok, use polymorphic destructor
  }
  virtual ~Swapable() { }
};
struct Orderable : virtual Comparable, virtual Swapable { // multiple virtual inheritance
  virtual ~Orderable() { }
};
```

☞ *simplicity 2/5, flexibility 3/5, extensibility 4/5*

Application of Design Principles *(concretions)*


```
class Integer : public Orderable {
public:
  Integer(int val_) : val(val_) { }
  virtual ~Integer() { }
  // overloaded methods (monomorphic)
  int  compare(Integer& i) { return val - i.val;   }
  void copy   (Integer& i) { val = i.val;          }
  void swap   (Integer& i) { std::swap(val,i.val); }
  // overridden methods (polymorphic)
  virtual Integer* clone() {                     // covariant return type (was Clonable*)
    return new Integer(*this);                   // ok, use default copy constructor
  }
private:
  virtual int compare(Comparable& c) {
    Integer& i = dynamic_cast<Integer&>(c);  // downcast, may raise an exception
    return this->compare(i);                 // this is covariant, no downcast
  }
  virtual void copy(Copiable& c) {
    Integer& i = dynamic_cast<Integer&>(c);  // downcast, may raise an exception
    this->copy(i);                           // this is covariant, no downcast
  }
  virtual void swap(Swapable& s) { // more efficient?
    Integer& i = dynamic_cast<Integer&>(s);  // downcast, may raise an exception
    this->swap(i);                           // this is covariant, no downcast
  }
  int val;
};
```

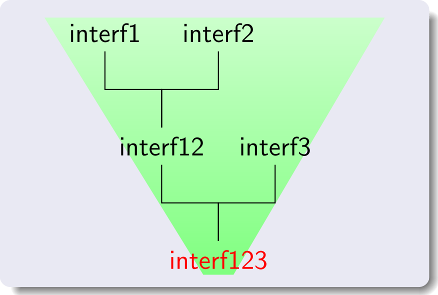
Application of Design Principles *(dependencies)*


```
void swap(Swapable& a, Swapable& b) {  // generic swap, not efficient
  a.swap(b); // polymorphic swap
}

void swap(Integer& a, Integer& b) {  // specific swap (overload), efficient
  a.swap(b); // monomorphic swap
}

int main() {
  Integer a(1), b(2);
  swap(a,b); // a.val == 2, b.val == 1
}
```

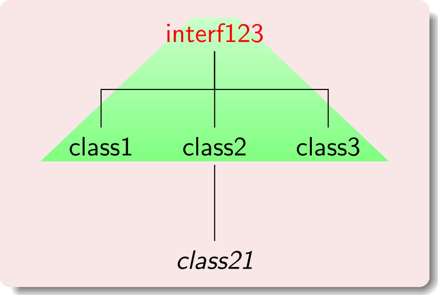
- ☞ *Interfaces can bloat objects sizes in some OO languages*
  - *instances of* Integer *should be* $\times 3$ *larger than* int
- ☞ *Parametric types (templates) suit better for small objects*
  - *designing parametric types is not easy (static duck typing)*
- ☞ *Multiple inheritance requires virtual inheritance (or member renaming)*

Dependency Inversion Principle

interfaces (abstract)



classes (concrete)



- ☞ *Interfaces enlarge static visibility with multiple inheritance*
- ☞ *Classes narrow static visibility with single inheritance*
- ☞ *Classes are for creation (RAII), interfaces are for use (algorithms)*

✌ *Dynamic typing does not care about static visibility* (no interface, flat hierarchy)

Object-Oriented Programming Techniques Summary

Summary

- Interfaces are an artefact of static typing ☹
- Interfaces provide a coherent abstraction of services ☺
- Interfaces provide flexibility by decoupling the interface from the implementation (encapsulation) ☺
- Interfaces provide extensibility by allowing new types to implement the services (polymorphism) ☺
- *Downcast enlarge static visibility (run-time type checking)* ☹
- *Covariance enlarge static visibility (compile-time type checking)* ☺
- *Interfaces have closed definitions and must stay small and orthogonal to be reusable* ☹
- *Multiple inheritance allows to compose orthogonal interfaces into larger interfaces suiting better to user-specific tasks* ☺
- *Dynamic typing does not create static dependencies* ☺☺☺

Outline

Final Advices

design

- Master your programming language before starting large projects
- Enforce abstraction and encapsulation
- Prefer expressive concepts (if available) to design patterns
- Prefer dynamic (aggregation) to static (inheritance) relationship

techniques

- Favor small interfaces and abstract types (object-oriented style)
  composed them with multiple inheritance
- Favor immutability and composition of closures (functional style)
- Avoid multiple inheritance on concrete types
- Avoid iterators (intrusive)
  Prefer mapping of functors/closures (non-intrusive)
- Avoid getters/setters (intrusive and destructuring)
  Prefer initialization and copy (non-intrusive)

Epilogue

It's hard to decode the Matrix

Are you ready to take the red pill?

Further Readings *(Books)*

- E. Gamma, R. Helm, R. Johnson, J. Vlissides
  Design Patterns: Elements of Reusable Object-Oriented Software
  Addison Wesley, 1995.
- J. Lakos
  Large-Scale C++ Software Design
  Addison Wesley, 1996.
- T. Khüne
  A Functional Pattern System for Object-Oriented Design
  Thesis, 1998.
- H. Sutter
  Exceptional C++
  Addison Wesley, 2000.
- S. Meyers
  Effective C++
  Addison Wesley, 2005.

Further Readings *(Articles)*

R.C. Martin
Design Principles and Design Patterns
ObjectMentor (http://www.objectmentor.com), 2000.

J. Bloch
How to Design a Good API and Why it Matters
LCSD'05 (http://lcsd05.cs.tamu.edu), 2005.

F. Steimann and P. Mayer
Patterns of Interface-Based Programming
Journal of Object Technology (http://www.jot.fm), 2005.