Dedicated Operating Systems

Operating System Engineering

LISHA/UFSC

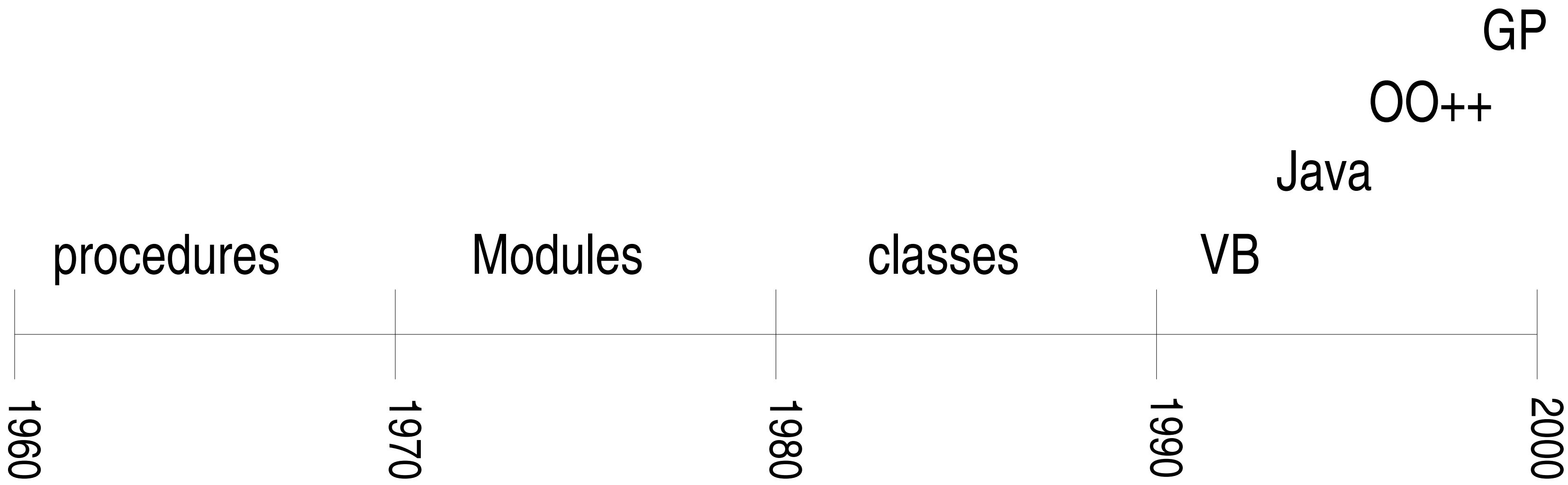
Prof. Dr. Antônio Augusto Fröhlich

guto@lisha.ufsc.br

http://www.lisha.ufsc.br/~guto

March 2004

Software Component Evolution



Software Component Definitions

"logically cohesive, loosely coupled module that denotes a single abstraction."

(Grady Booch, 1987)

"already implemented units that we use to en-hance the programming language constructs."

(Ivar Jacobson, 1993)

"self-contained, clearly identifiable pieces that describe and/or perform specific functions, have clear interfaces, appropriate documentation, and a defined reuse status."

(Johannes Sametinger, 1997)

"unit of composition with contractually specified interfaces and explicit context dependencies only."

(Clemens Szyperski, 1997)

Software Component Granularity

- High granularity
  - Specialized
  - Reusable
  - Efficient
- Low granularity
  - Generic
  - Composable

e.g. subsystem
Do I have to take this junk as well?
The component I need is not there!
a few large components

component granularity

a lot of small components
What is this component for?
What is the difference to that other one?
How do I use it?
e.g. container

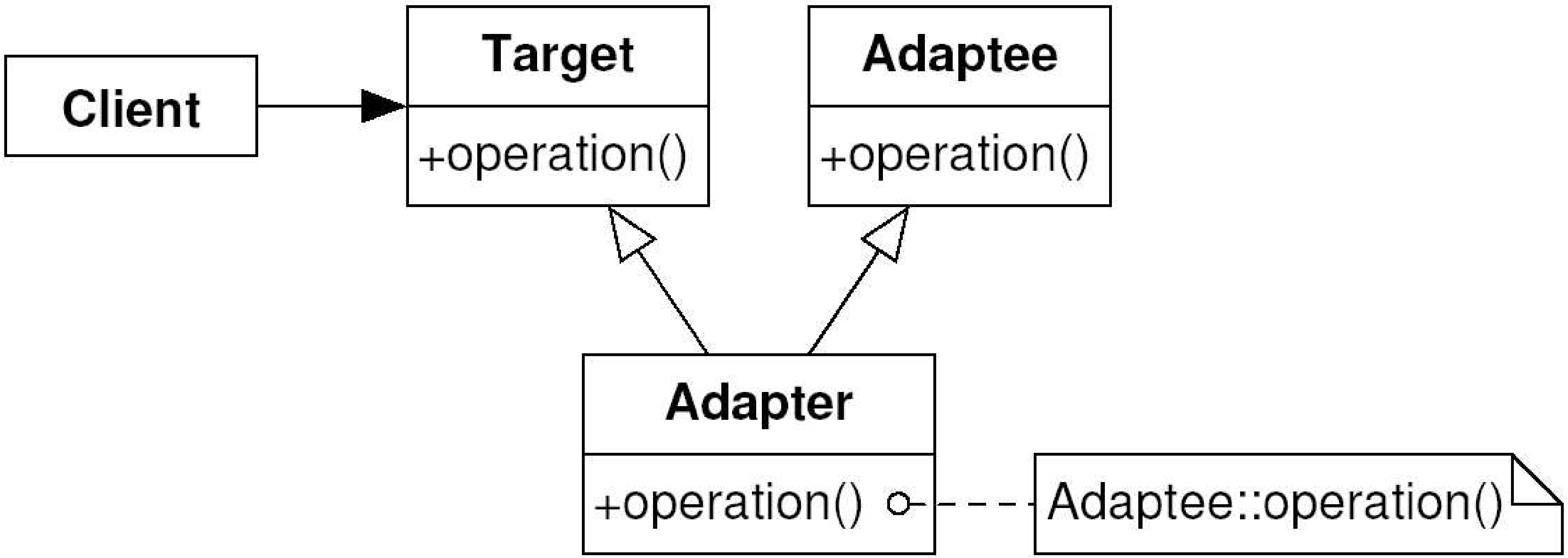
Software Component Interfaces

- Service contracts
  - Clients: what to expect and how to deploy
  - Providers: what to implement
- Formal contracts
  - Syntax: interface
  - Behavior: pre and post conditions ...
  - Support composition validation

Design Patterns

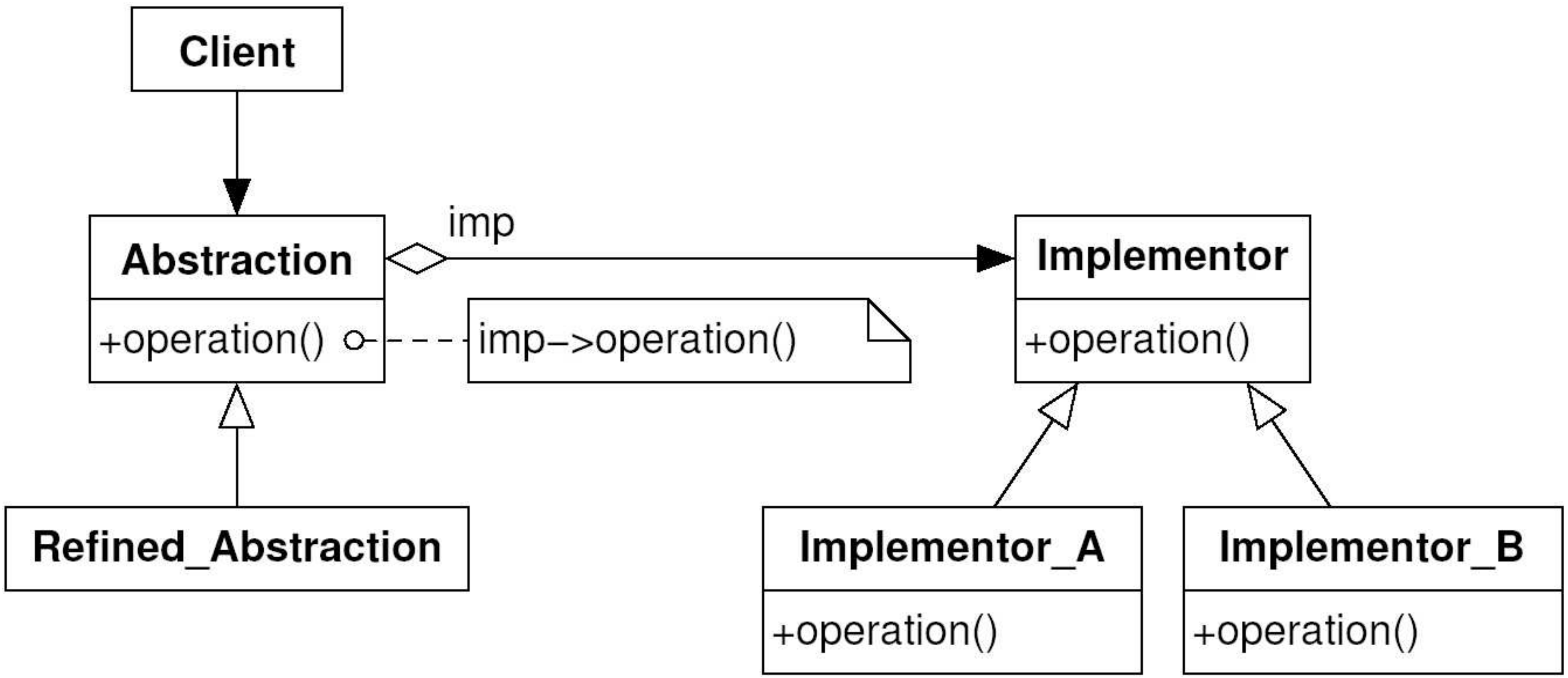
- Catalog of solutions to recurring problems in the object-orientation scene
  - Taxonomy of elementary OO architectures
- Orthogonal to domain matters
- Composition
  - Adapter: incompatible interfaces
  - Bridge: decouple abstraction and implementation

The Adapter Pattern



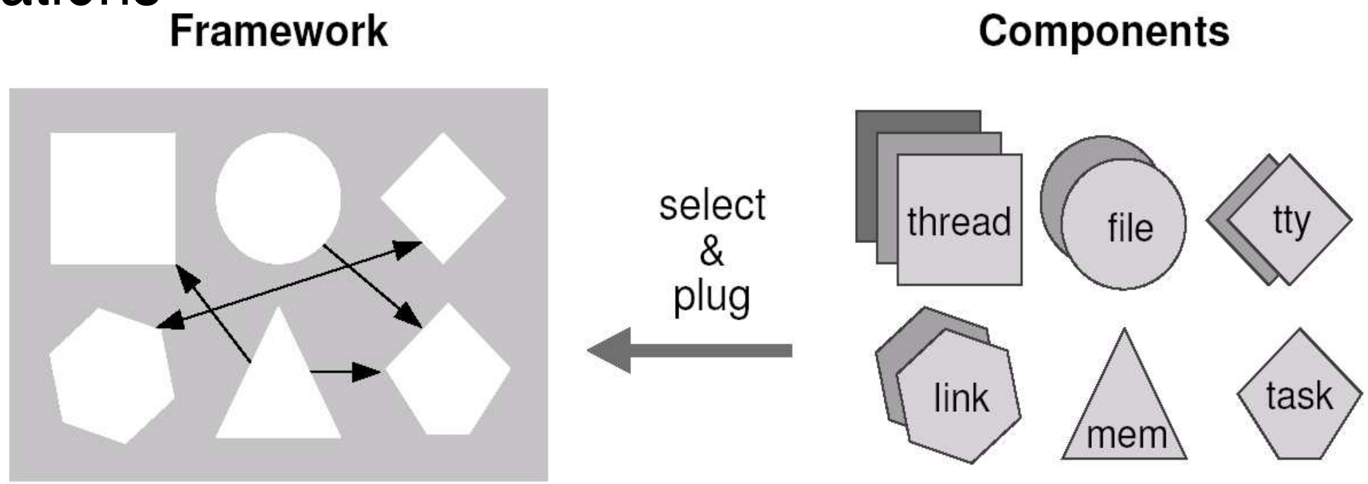
Dedicated Operating Systems

The Bridge Pattern



Frameworks

- Arrangement of classes that captures a reusable design
  - Abstract: implementation inheritance
  - Concrete: reusable implementations
- Whitebox framework
  - Inheritance and overriding
- Blackbox (component framework)
  - Interfaces and composition
- System-wide properties

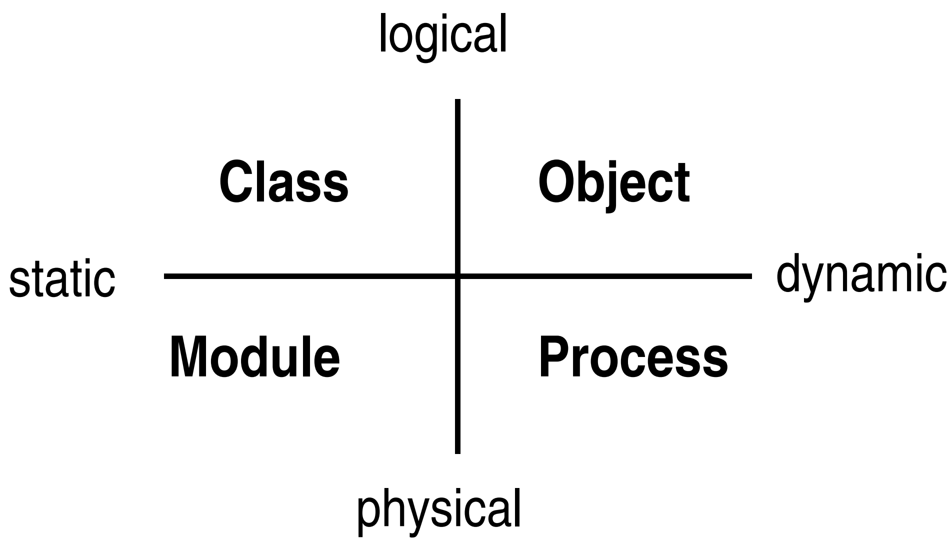


Family-Based Design

- Commonality and variability analysis
  - Commonality => families
  - Variability => family members
- Incremental system design
  - Hierarchy in family-based design
- Family-oriented abstraction, specification, and translation
  - Application-oriented languages (AOL) to hide commonalities as design secrets

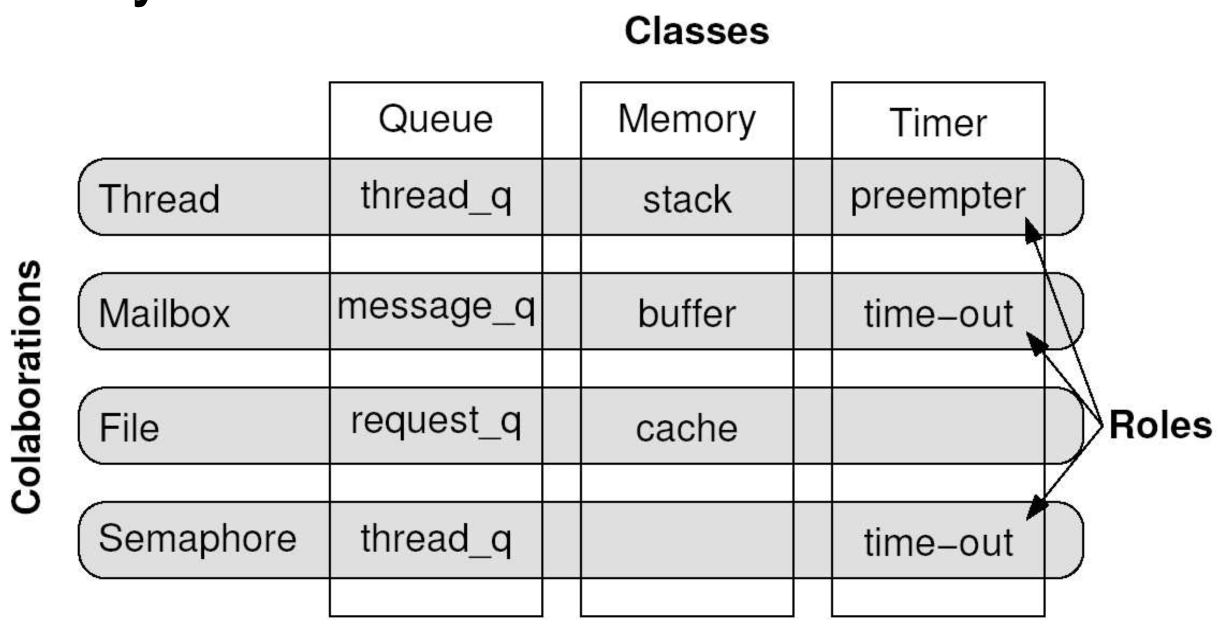
Object-Oriented Design

- Domain analysis and decomposition
  - Objects abstract domain entities
  - Commonality => classes
  - Variability => class hierarchies (subclassing)
- Models



Collaboration-Based Design

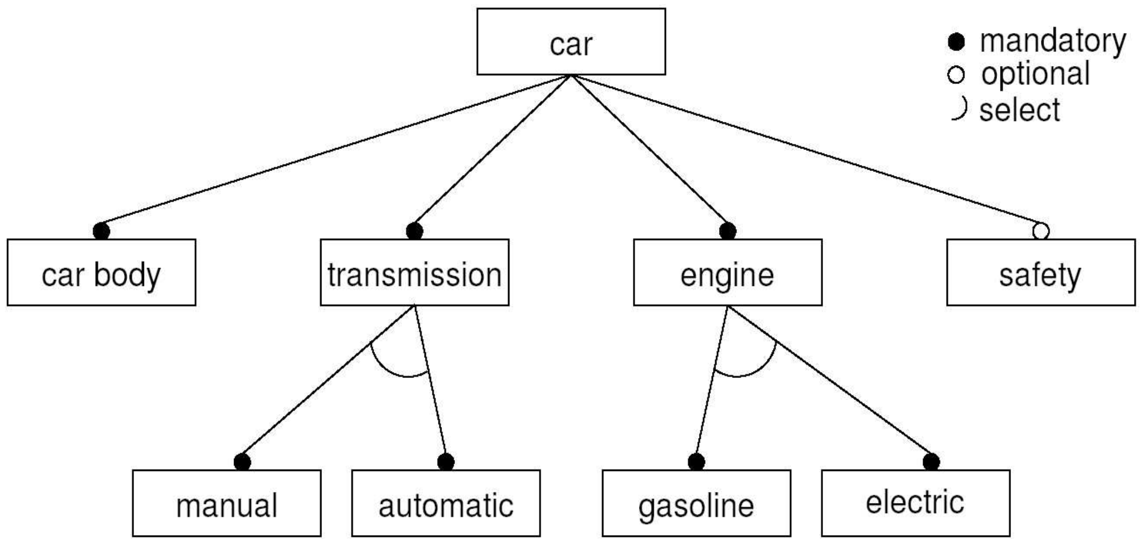
- Extends object-oriented design
  - An object can play different roles in a system
  - A cooperating suite of roles (collaboration) can be a better unit of reuse than a class
- Collaboration-based system
  - Composition of independently definable collaborations



Feature-Based Modeling

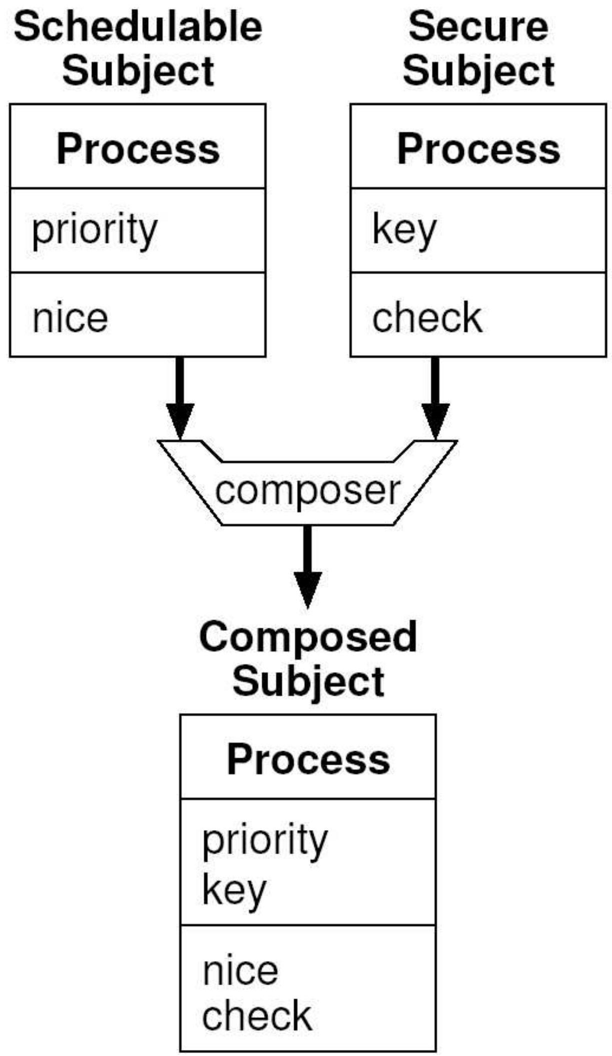
Dedicated Operating Systems

- **Features** enable the design process to be approached from **varying levels of detail**
  - Sub-features provide a method for viewing features as an aggregation of several, more primitive features
- Natural to use
  - Structures, behaviors, and names are **recognizable by designers**
- Feature-Oriented Domain Analysis (FODA)



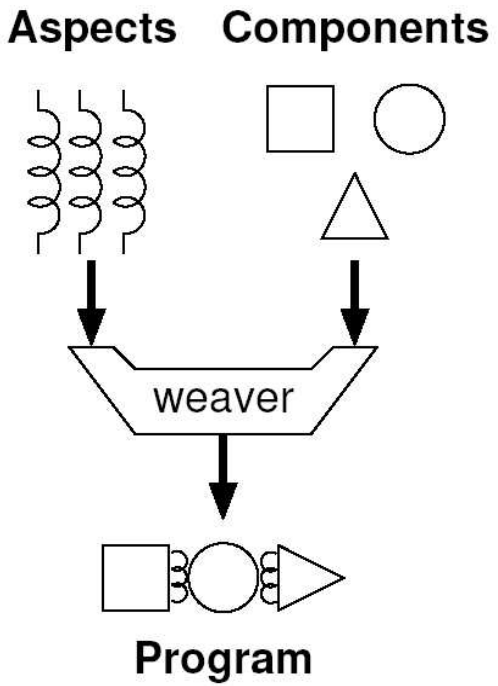
Subject-Oriented Programming

- Extends object-orientation to handle a multiplicity of subjective views of objects been modeled
  - Some properties of an object can be more interesting to some programs than to others
- Subjects: model subjective view of domain
- Subject composition: reconcile subjective views



Aspect-Oriented Programming

- Deals with non-functional properties of component-based systems
  - Replace code fragments scattered over several components with reusable aspects
- Aspects
  - Specified in aspect-oriented languages
  - Woven with components



Aspect-Oriented Programming Example

Dedicated Operating Systems

```
aspect Action
{
  advice execution("% A::%(...)") : around() {
    cout << "before exec " << JoinPoint::signature();
    cout << "[that=" << (void *)tjp->that() << ",";
    cout << "target=" << (void *)tjp->target() << "]\n";
    tjp->proceed();
    cout << "after exec " << JoinPoint::signature() << "\n";
  }
  advice call("% A::%(...)") : around() {
    cout << "before call " << JoinPoint::signature();
    cout << "[that=" << (void *)tjp->that() << ",";
    cout << "target=" << (void *)tjp->target() << "]\n";
    tjp->proceed();
    cout << "after call " << JoinPoint::signature() << "\n";
  }
};
```

OUTPUT:

```
before call int A::a(int,float) [that=(nil), target=0xbfffed0f]
before exec int A::a(int,float) [that=0xbfffed0f, target=0xbfffed0f]
after exec int A::a(int,float)
after call int A::a(int,float)
```

Generic Programming

- Reusability by means of parameterization
  - Decouple algorithms from data structures
- Generic components
  - Externally adjustable (parameters)
  - Compile-time
- C++ Standard Template Library (STL)

Generic Programming Example


```
template <int n_res, class Resource>
class Allocator
{
public:
  Allocator() { for (int i = 0; i < n_res; i++) used[i] = false; }
  Resource* alloc() {
    int i;
    for (i = 0; (i < n_res) && used[i]; i++);
    return (i == n_res) ? 0 : (used[i] = true, &resource[i]);
  }
  void free(Resource * res) {
    int i;
    for (i = 0; (i < n_res) && (&resource[i] != res); i++);
    if (i != n_res) used[i] = false;
  }
private:
  bool used[n_res];
  Resource resource[n_res];
};
```

Static Metaprogramming

- Multilevel languages
  - Parts of the input program are evaluated at compile-time
  - Supported by C++
    - Templates, expression evaluation, inlining
- Component transformation and composition

```
template <int n>
struct Factorial { enum { RET = Factorial<n - 1>::RET * n }; };

template <>
struct Factorial<0> { enum { RET = 1 }; };
```

Generative Programming

- Domain engineering
  - Families
- Configuration knowledge
  - Components into product
- Generators
  - Aspect-oriented programming
  - Subject-oriented programming
  - Static metaprogramming

Multiparadigm Design

- A single paradigm cannot cover peculiarities of all domains
  - Paradigms have to be combined
- Example
  - Object-orientation +
  - Family-based +
  - Structured +
  - Logic +
  - ...