Design Strategies

Jim Fawcett
CSE687 – Object Oriented Design
Spring 2015

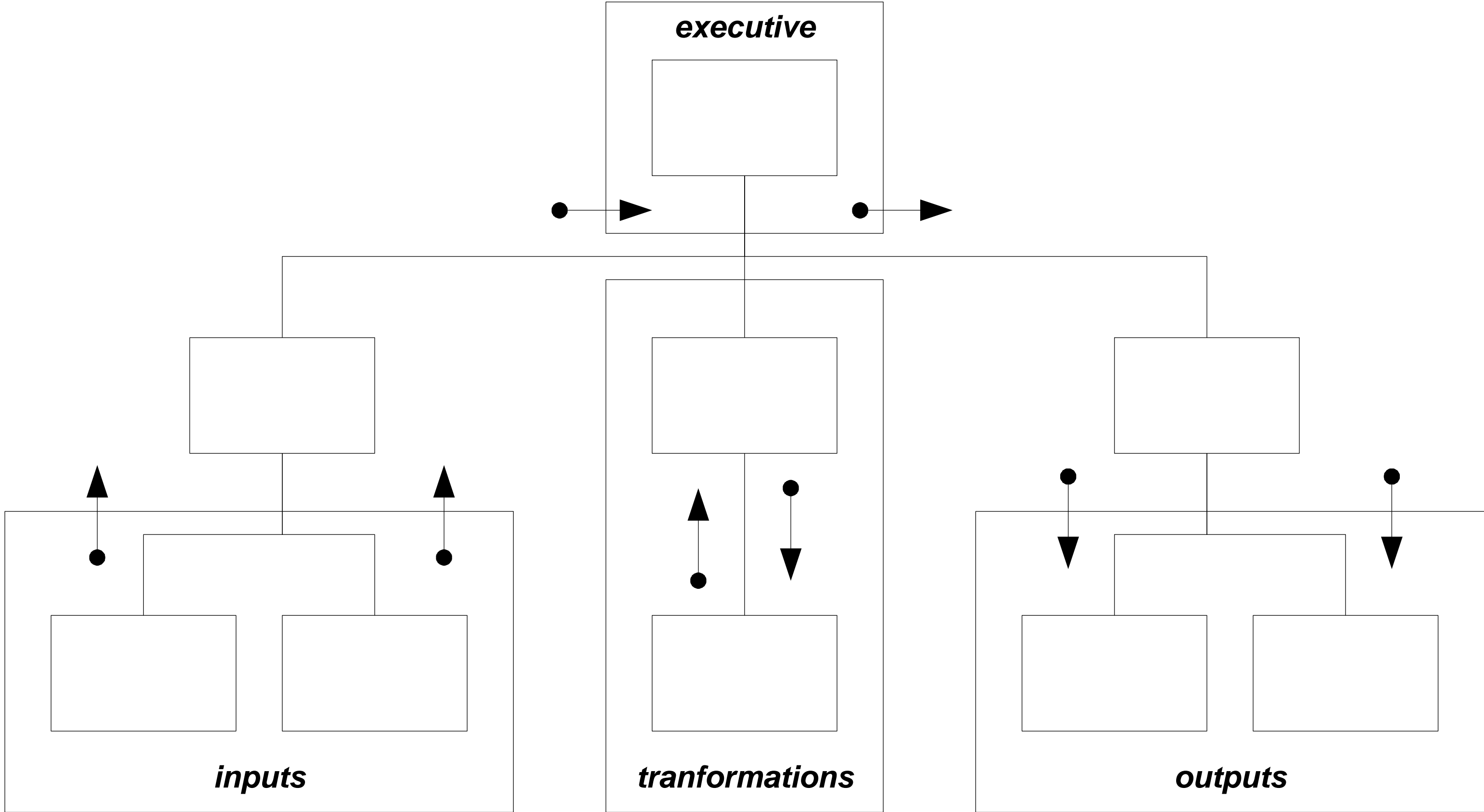
Strategies

- ***Top down design***
  - Task oriented decomposition
  - Uses successive refinement to define lower layers
- ***Bottoms up design***
  - Focus on encapsulating an abstraction
  - Provides a language for higher layers to use.
- ***Object oriented design***
  - Partitions program activities into object interactions
  - Defines objects as members of families using inheritance and composition.

Classical Structure Chart

- The terms “Top Down” and “Bottoms Up” refer to the classical form of a structure chart.
  - This form puts logical, executive, processing at the top and low level, physical, processing on the bottom.
  - The classical form puts inputs on the left side and outputs on the right, leading to terms like left-to-right design and right-to-left design.
  - Left-to-right design focuses on program input
  - Right-to-left design is concerned with program output.

Classical Structure Chart



Top Down Design

- Make task list
- Decompose tasks into packages
  - Executive interfaces and servers
- Proceed from top-level logical model to lower-level, more physical models
  - Process of successive refinement
  - Only effective for a very few layers

Top Down Design

- Advantages:
  - Focus on satisfying requirements.
  - Doesn't invent things not needed for this project

- Disadvantages:
  - Generates non-reusable parts
  - Makes tightly coupled structure

Bottoms Up Design

- Identify useful data structures and operations.
- Encapsulate and provide operations to manage them.
- Essentially the same process for either packages or classes.
- Usually start by specifying the package, then populating with classes.

Bottoms Up Design

- Advantages:
  - Makes reusable parts.
  - Abstractions you develop here makes higher level implementation much easier.
- Disadvantages:
  - May invent a lot of function-ality not needed for the project.
  - ***Starting*** with bottoms up design often leads to poorly structured over-all design.

Object Oriented Design

- Partitions project activities into interactions of class objects.
- Defines families of classes and their relationships:
  - Using
  - Aggregation
  - Composition
  - Inheritance
  - Template based pluggability
- Object Oriented Design is used during both top-down and bottoms-up design
  - Top-Down: define class hierarchies
  - Bottoms-Up: define each class's implementation

OOD Strategies

- Define an abstraction and encapsulate its implementation.
  - Very often centers on encapsulating a data structure and managing its transformations
- Layer implementation using composition.
- Extend an abstraction through inheritance.
- Loosely couple interacting objects using polymorphism.
  - Between programs with late design binding.
  - Within programs using run-time binding.

Binding

- To bind is to associate a reference to an object or global function with a specific implementation:
  - May statically bind by providing a name and type.
    - Early design time – conventional class design
    - Late design time – provide an application specific instantiation of a template-based class
  - May dynamically bind at run-time by creating an object on the heap bound through a pointer or C++ reference.
    - Pointers or C++ references are essential because we can provide names only at design time, not at run time.

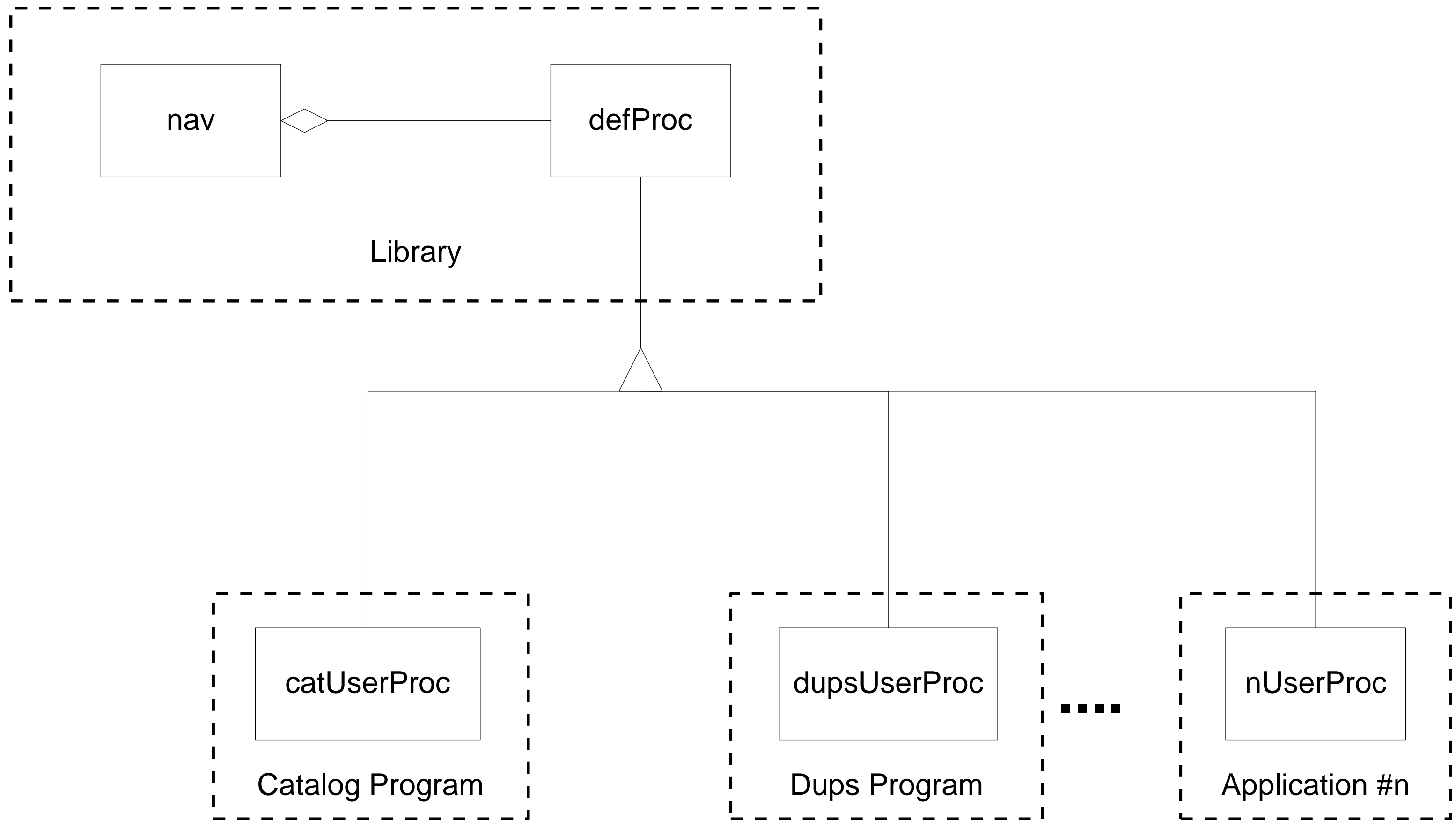
Template Example

- vector<T> is bound only when T is specified during the client's design.
  - This is late design time binding, e.g., after the design of the template class.
  - The STL library provides the vector<T> container type, unbound to any specific type (early design time).
  - Your design binds to a type, e.g., vector<int> vint;
    (late design time).

Inheritance Example

- The catalog program uses an instance of the navig class that communicates through a base class, defProc, pointer to a derived class, userProc, object.
  - The defProc class is an early design time artifact, developed before application requirements are known.
  - The userProc class is a late design time artifact, developed when the application requirements are known.
  - Explicit binding of userProc to navig occurs at run-time when executive registers userProc object with an instance of the navig class.

Inheritance Example



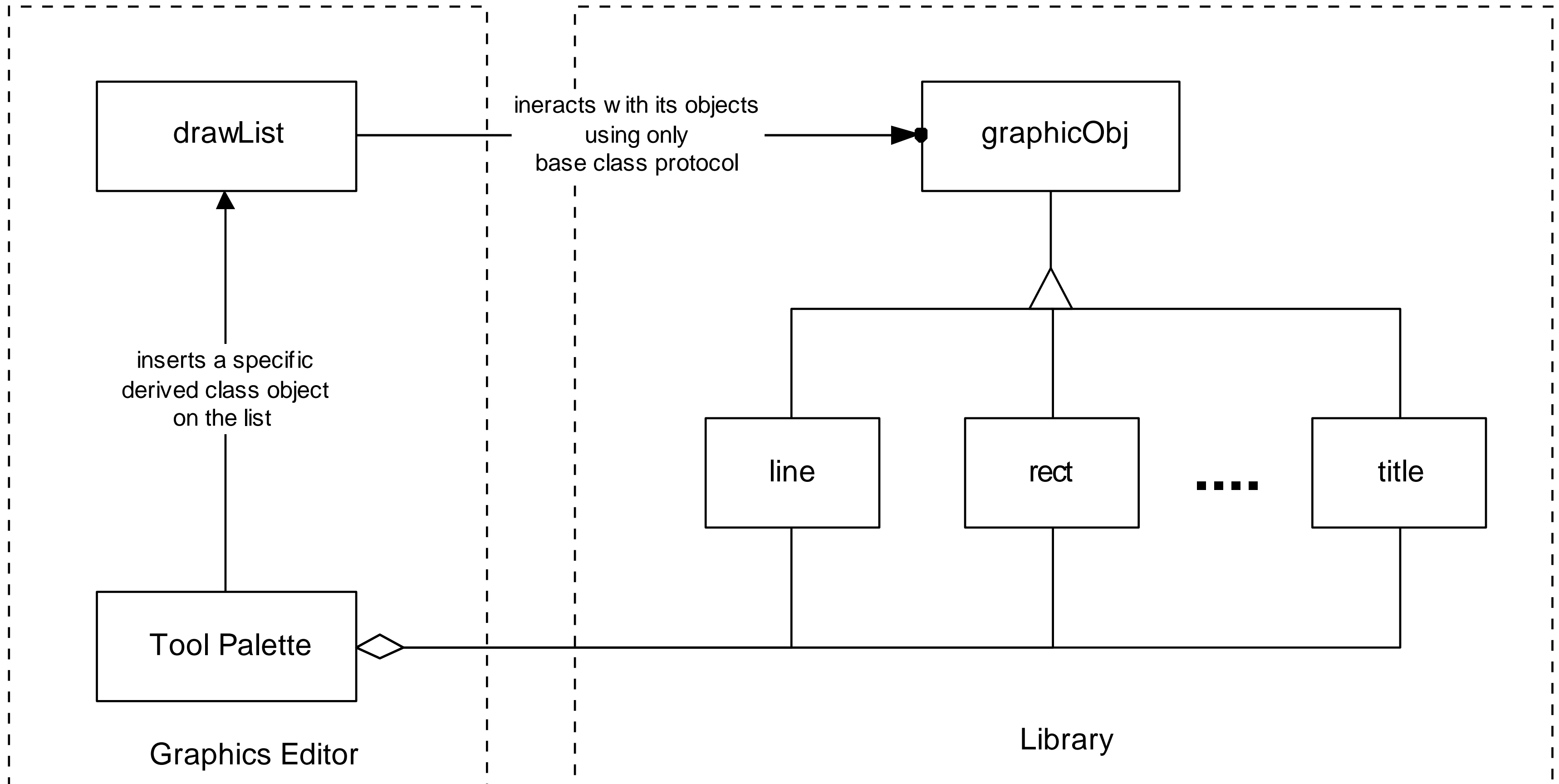
Inheritance Example

- Note that in this example we have a relatively unusual situation:
  - The library code makes calls into the application's code.
  - To do this with a reusable library component it is essential that binding is deferred until late design time, e.g., after the library is implemented.

Another Inheritance Example

- The graphics editor example uses run-time binding to loosely couple a client from the implementation being used.
- The drawlist client and all other clients of the graphicObj hierarchy use only the base class protocol provided by the graphicObj class to draw(), move(), or erase() derived objects.
- Thus, drawlist and lots of other code in the editor are blissfully ignorant of the details that distinguish one derived class from another, e.g., a line from a box from a title, etc.
- Only the tool palette needs to know the details that distinguish one derived class from another, in this case, simply because palette needs to create instances of specific classes to place on the editor's drawlist.

Graphics Editor Example



Graphics Editor Summary

- The drawList, and almost all the other components in the Graphics Editor use the library though its base class protocol, so they have no compile dependencies on the library. Should we add another derived type, their designs do not break, nor do we even have to recompile them.

- Very small parts of the Graphics Editor, like the Tool Palette, may need to know more, and bind more tightly, by using the specific names of the derived classes via a dynamic_cast operation. Such components will have design dependencies on the derived classes.

- Note that the Tool Palette acts as an object factory for the other Graphics Editor components. It is, in fact, the use of protocol classes ***and*** object factories that result in compile independence between clients and the graphicObj hierarchy.

OOD Strategies Revisited

- Encapsulate the implementation of an abstraction with a class.
- Layer implementation using composition.
- Extend an abstraction through inheritance.
- Loosely couple interacting objects using polymorphism.
  - Between programs with late design binding.
  - Within programs using run-time binding.

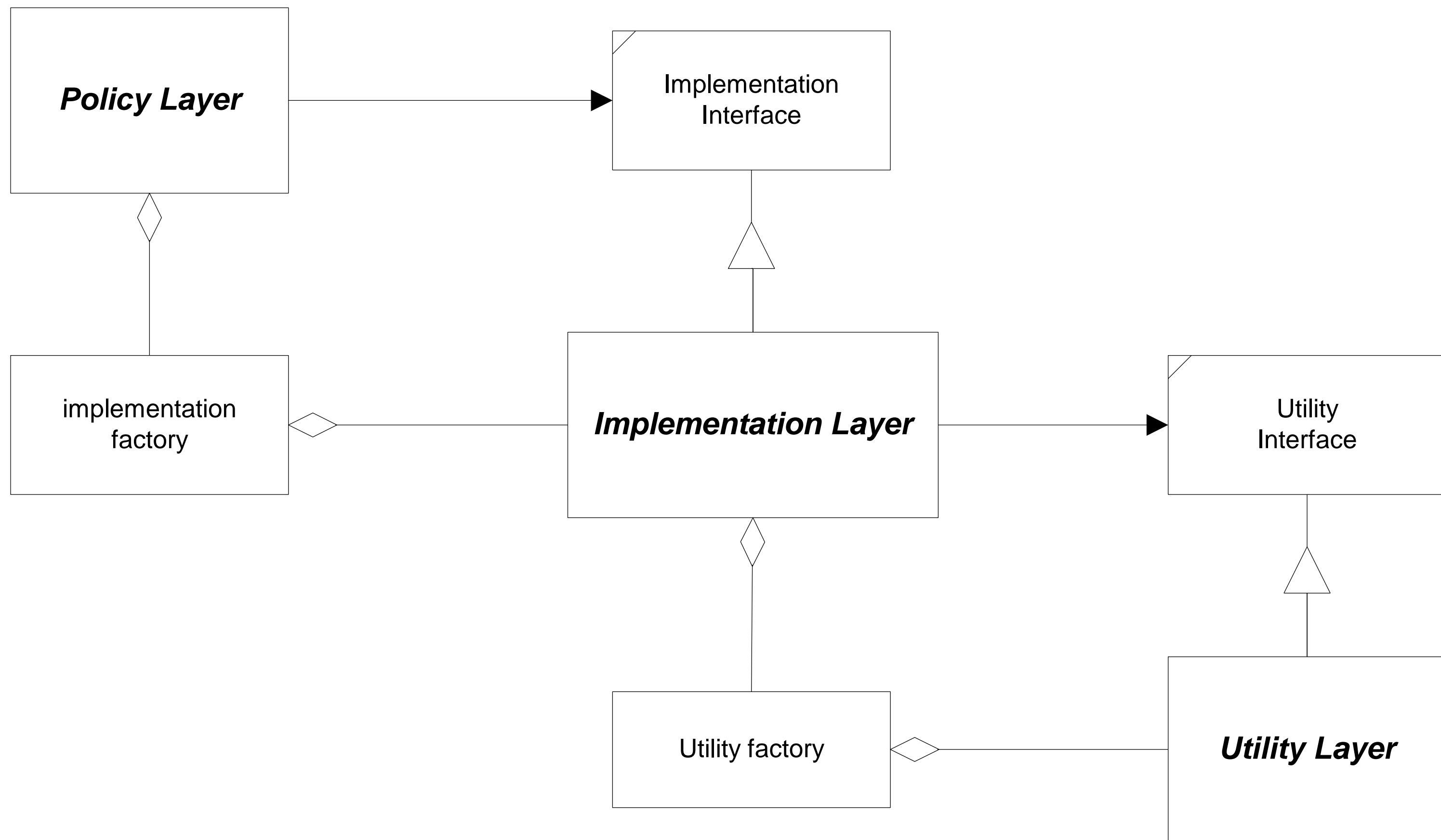
OOD Strategies

- Encapsulation and composition are essentially bottoms up design tools.
- Using inheritance and polymorphism are not. They are something new and powerful, unique to OOD.

OOD Principles

- We will discuss three principles that all focus on inheritance and late binding:
  - Open/Closed Principle says that we should not modify objects, we extend them, as in the catalog program example.
  - Liskov Subsitution Principle says that we must design class families so that derived objects have the syntax and semantics to be used wherever base class objects are expected.
  - Dependency Inversion Principle says that we should separate large volatile partitions with abstract interfaces (which uses inheritance).

Breaking Dependency



Design Layers

- It is very typical that the OOD strategies map in different ways onto the three layers shown on the previous slide.
  - Top down design determines a policy layer and top-level partitions of the implementation layer – very little OOD at this level. Here we use modules to partition responsibilities.
  - Classes and class relationships dominate the implementation layer. Polymorphism is an important tool to minimize coupling between components of this layer and with the utility layer.
  - Class encapsulation of data and operating system services, using bottoms up design, determines the utility layer.

OOD Strategies Revisited

- Encapsulate the implementation of an abstraction with a class.
- Layer implementation using composition.
- Extend an abstraction through inheritance.
- Loosely couple interacting objects using polymorphism.
  - Between programs with late design binding.
  - Within programs using run-time binding.

Parting Comments

- Try not to bind to things that change:
  - Platform specific code
  - Code in the midst of development

- Early design (potential library) must try not to bind to late design (application specific details)
  - Otherwise the early designs can never be reusable.
  - Satisfying this rule without polymorphism or templates would force all reusable design to be bottoms up.

End of Presentation