COMP 3700: Software Modeling and Design

(Creational Design Patterns)

Agenda

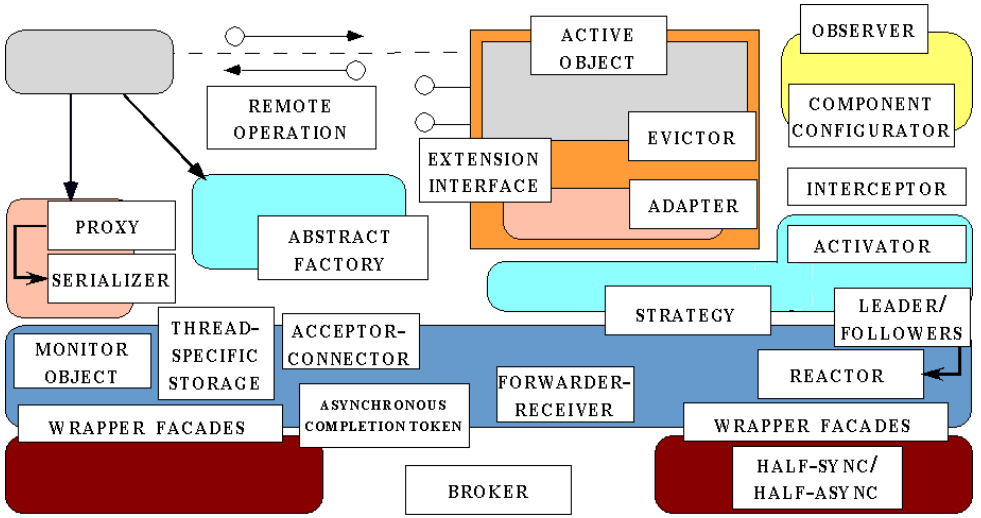
- Introduction to Patterns
- Creational Patterns
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton

Patterns

- Present *solutions* to common software *problems* arising within a certain *context*
- Capture recurring structures & dynamics among software participants to facilitate reuse of successful designs

Pattern Languages

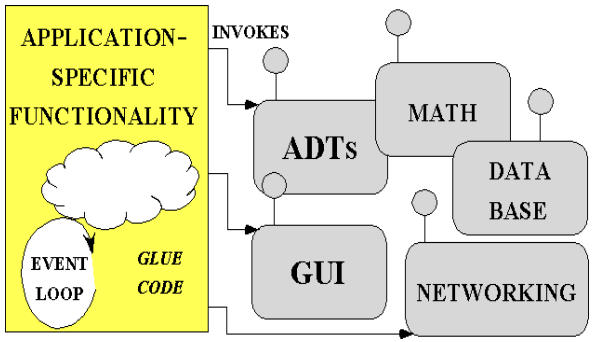
- Define a *vocabulary* for talking about software development problems
- Provide a *process* for the orderly resolution of these problems
- Help to generate & reuse software *architectures*



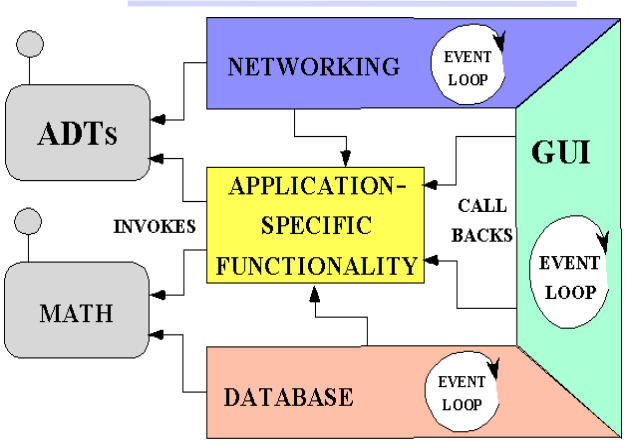
Frameworks

- *An integrated collection of components that collaborate to produce a reusable architecture for a family of related applications*

Frameworks	Class Libraries
"Semi-complete" applications	Stand-alone components
Domain-specific	Domain-independent
Inversion of control	Borrow caller's thread of control

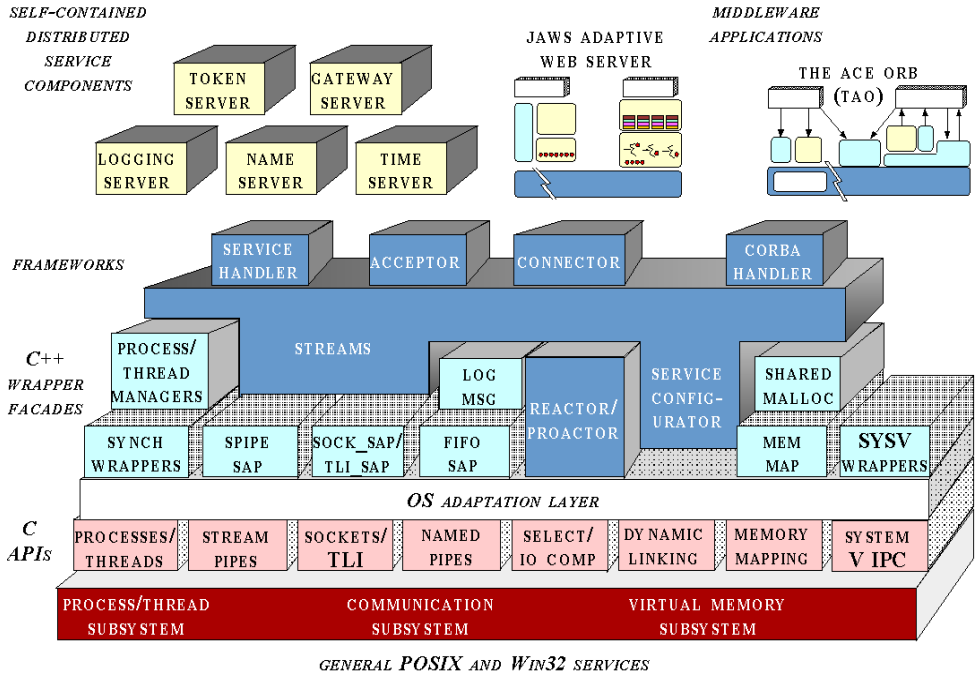


Class Library Architecture



Framework Architecture

An Example: ACE FRAMEWORK



- **Large open-source user community**
  - www.cs.wustl.edu/~schmidt/ACE-users.html

Creational Patterns

Abstract Factory

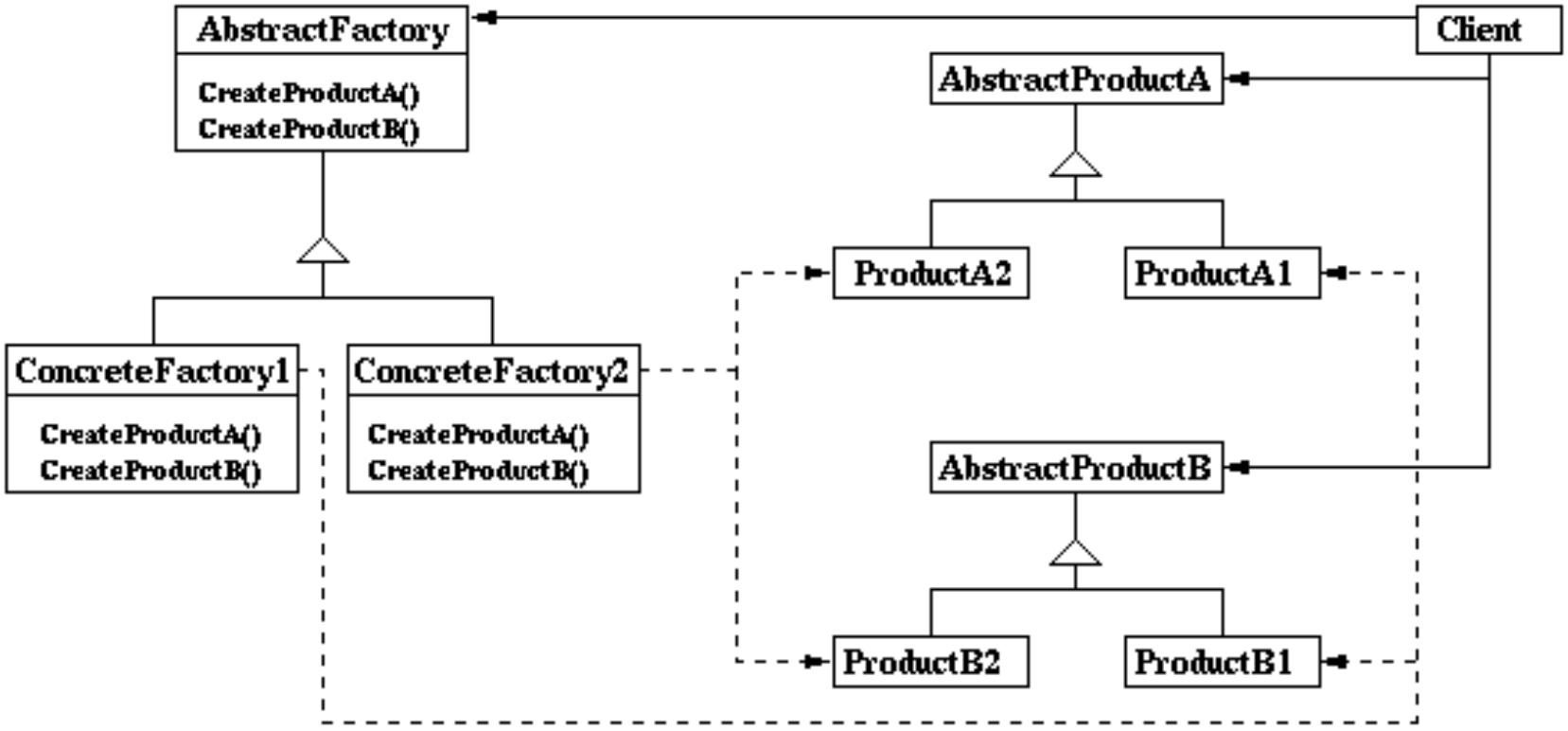
Intent: Provide an interface for creating families of related objects.

Applicability: Use the Abstract Factory pattern when

- a system should be independent of how its products are created, composed, and represented.
- a system should be configured with one of multiple families of products.
- a family of related product objects is designed to be used together, and you need to enforce this constraint.
- you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Abstract Factory

Structure



Abstract Factory

Participants

AbstractFactory
declares an interface for operations that create abstract product objects.
ConcreteFactory
implements the operations to create concrete product objects.
AbstractProduct
declares an interface for a type of product object.
ConcreteProduct
defines a product object to be created by the corresponding concrete factory.
implements the AbstractProduct interface.

Consequences

The Abstract Factory pattern has the following benefits and liabilities:

1. It isolates concrete classes.
2. It makes exchanging product families easy.
3. It promotes consistency among products.
4. Supporting new kinds of products is difficult

Builder

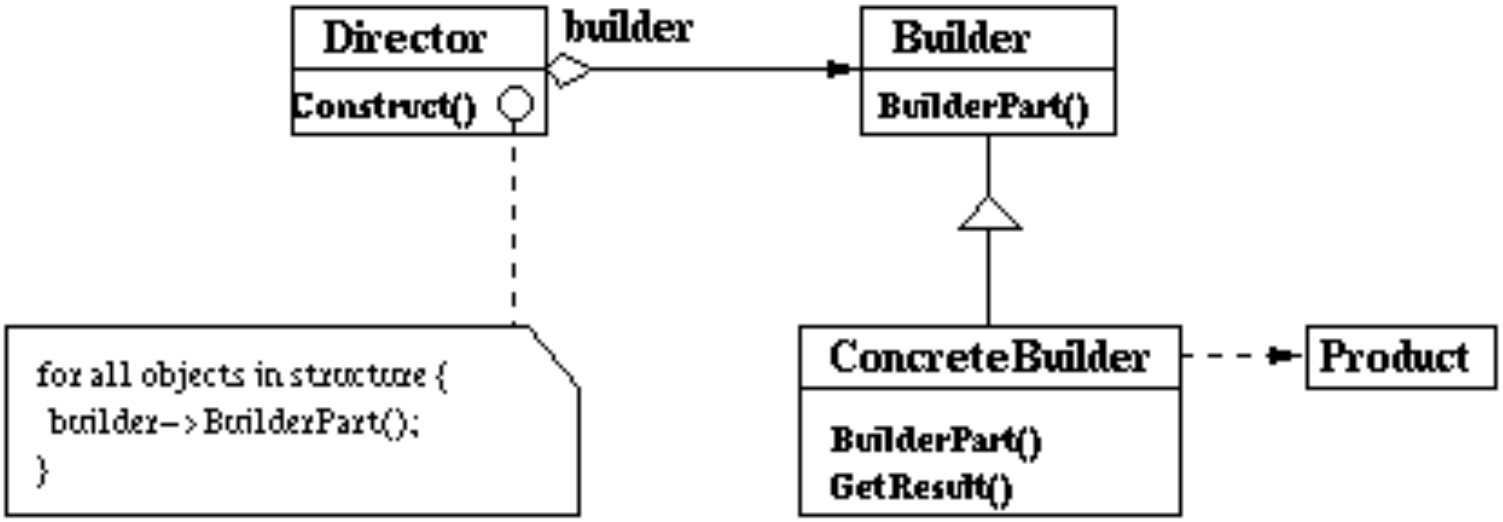
Intent : Create composite objects of different representations.

Applicability: Use the builder pattern when

- **the algorithm for creating a complex object should be independent of the parts that makes up the object and how they assembled.**
- **the construction process must allow different representations for the object that's constructed.**

Builder

Structure



Builder

Participants

Builder

specifies an abstract interface for creating parts of a Product object.

ConcreteBuilder

constructs and assembles parts of the product by implementing the Builder interface. defines and keeps track of the representation it creates. provides an interface for retrieving the product.

Director

constructs an object using the Builder interface.

Product

represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled. includes classes that define the constituent parts, including interfaces for assembling the parts into final result.

Consequences

1. It lets you vary a product's internal representation.
2. It isolates code for construction and representation.
3. It gives you finer control over the construction process.

Factory Method

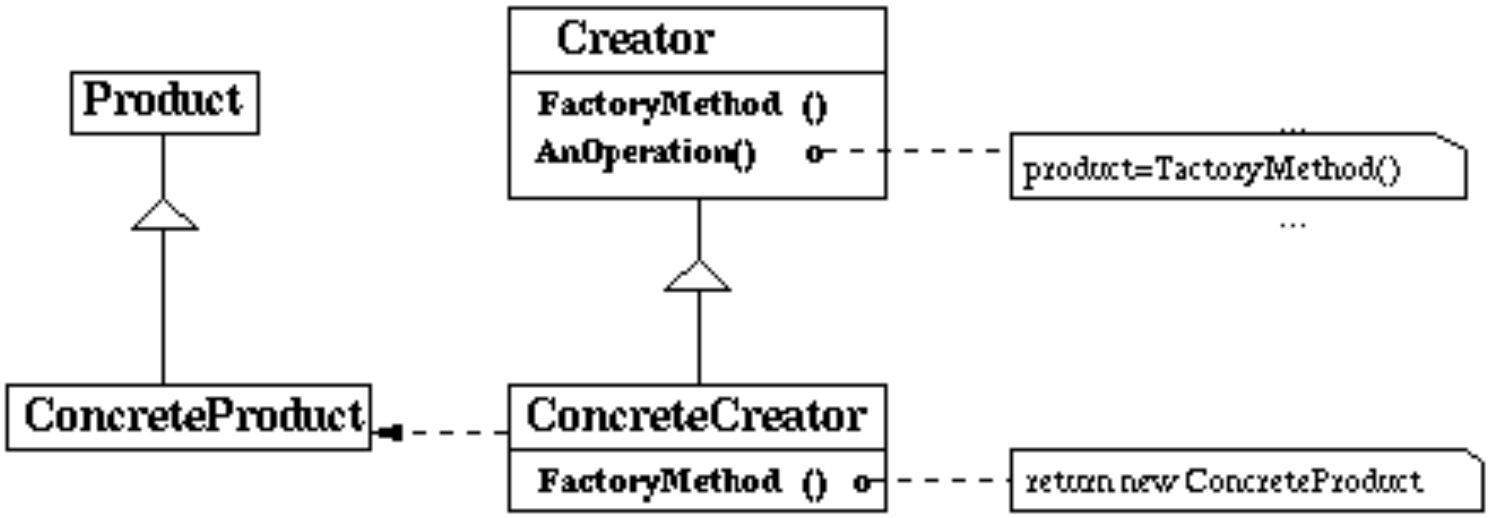
Intent : Define an interface for creating an object, but defer instantiation to subclasses.

Applicability : Use the Factory Method pattern when

- **a class can't anticipate the class of objects it must create.**
- **a class wants its subclasses to specify the objects it creates.**
- **classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper**
- **subclass is the delegate.**

Factory Method

Structure



Factory Method

Participants

Product
defines the interface objects the factory method creates.

ConcreteProduct
implements the Product interface.

Creator
declares the factory method, which returns an object of type Product.
Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
may call the factory method to create a Product object.

ConcreteCreator
overrides the factory method to return an instance of a ConcreteProduct.

Consequences

It provides hooks for subclassing.
It connects parallel class hierarchies.

Prototype

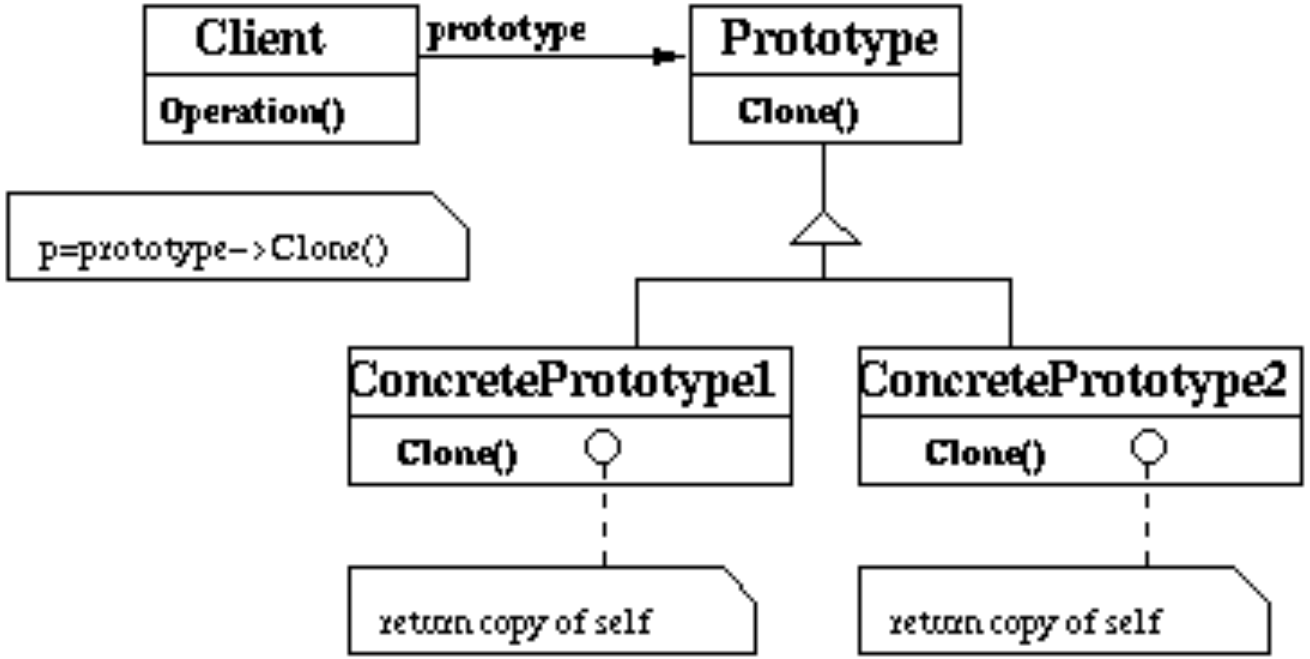
Intent : Create new objects by copying a prototype.

Applicability : Use the Prototype when

- **a system should be independent of how its products are created, composed, and represented; and**
- **when the classes to instantiate are specified at run-time, for example, by dynamic loading; or**
- **to avoid building a class hierarchy of factories that parallels the class hierarchy of products; or**
- **when instances of a class can have one of only a few different combinations of states.**

Prototype

Structure



Prototype

Participants

Prototype
declares an interface for cloning itself.

ConcretePrototype
implements an operation for cloning itself.

Client
creates a new object by asking a prototype to clone itself.

Consequences

1. It hides the concrete product classes from the client.
2. It let client work with application-specific classes without modification.
3. It adds and removes products at run-time.
4. It specifies new objects by varying values.
5. It specifies new objects by varying structures.
6. It reduces subclassing.
7. It configures an application with classes dynamically.

Singleton

Intent

Ensure a class has only one instance.

Applicability

Use the singleton pattern when

- there must be exactly one instance of a class, and it must be accessible to client from a well-known access point.
- when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Singleton

Participants

Singleton defines an Instance operation that lets clients access its unique instance. Instance is a class operation. may be responsible for creating its own unique instance.

Consequences

Singleton has following benefits:

1. Controlled access to sole instance.
2. Reduced name space.
3. Permits refinement of operations and representation.
4. Permits a variable number of instances.
5. More flexible than class operations.

Structure