What is a Design Pattern

„Each pattern describes a problem which occurs
over and over again in our environment,

and then describes the core of the solution to that problem,

in such a way that you can use this solution
a million times over,

without ever doing it the same way twice“

(Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel, “A Pattern Language: Towns/Buildings/ Construction”, Oxford University Press, New York, 1977)

Elements of Design Patterns

- Pattern Name
  - Increases design vocabulary, higher level of abstraction
- Problem
  - When to apply the pattern
  - Problem and context, conditions for applicability of pattern
- Solution
  - Relationships, responsibilities, and collaborations of design elements
  - Not any concrete design or implementation, rather a template
- Consequences
  - Results and trade-offs of applying the pattern
  - Space and time trade-offs, reusability, extensibility, portability

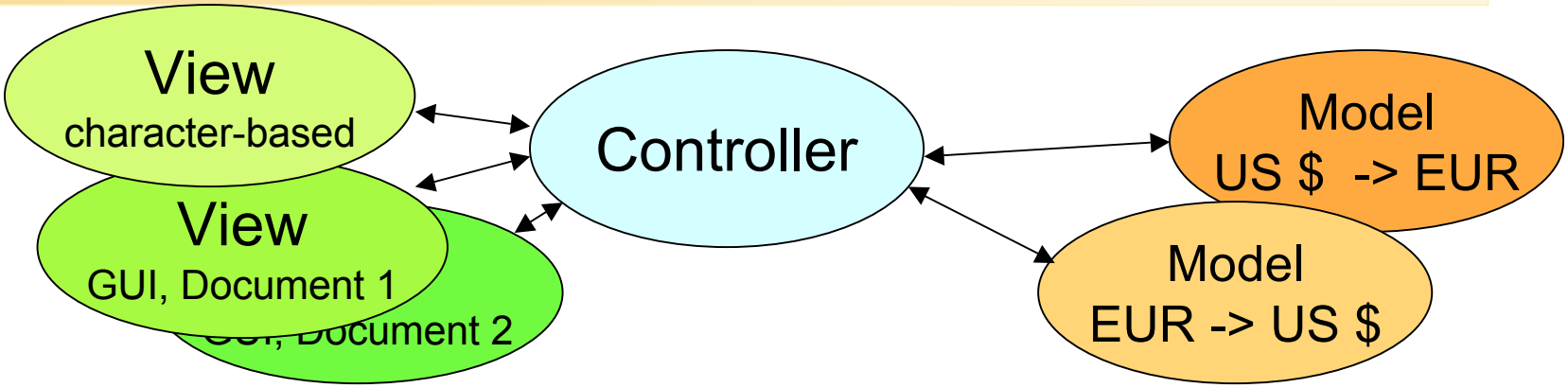
What is a Design Pattern (II)

- Description of communicating objects and classes that are customized to solve a general design problem in a particular context.

  (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, “Design Patterns – Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1994 (22nd printing July 2001))

- Each pattern focuses in a particular object-oriented design problem or issue

Design Patterns in Smalltalk MVC

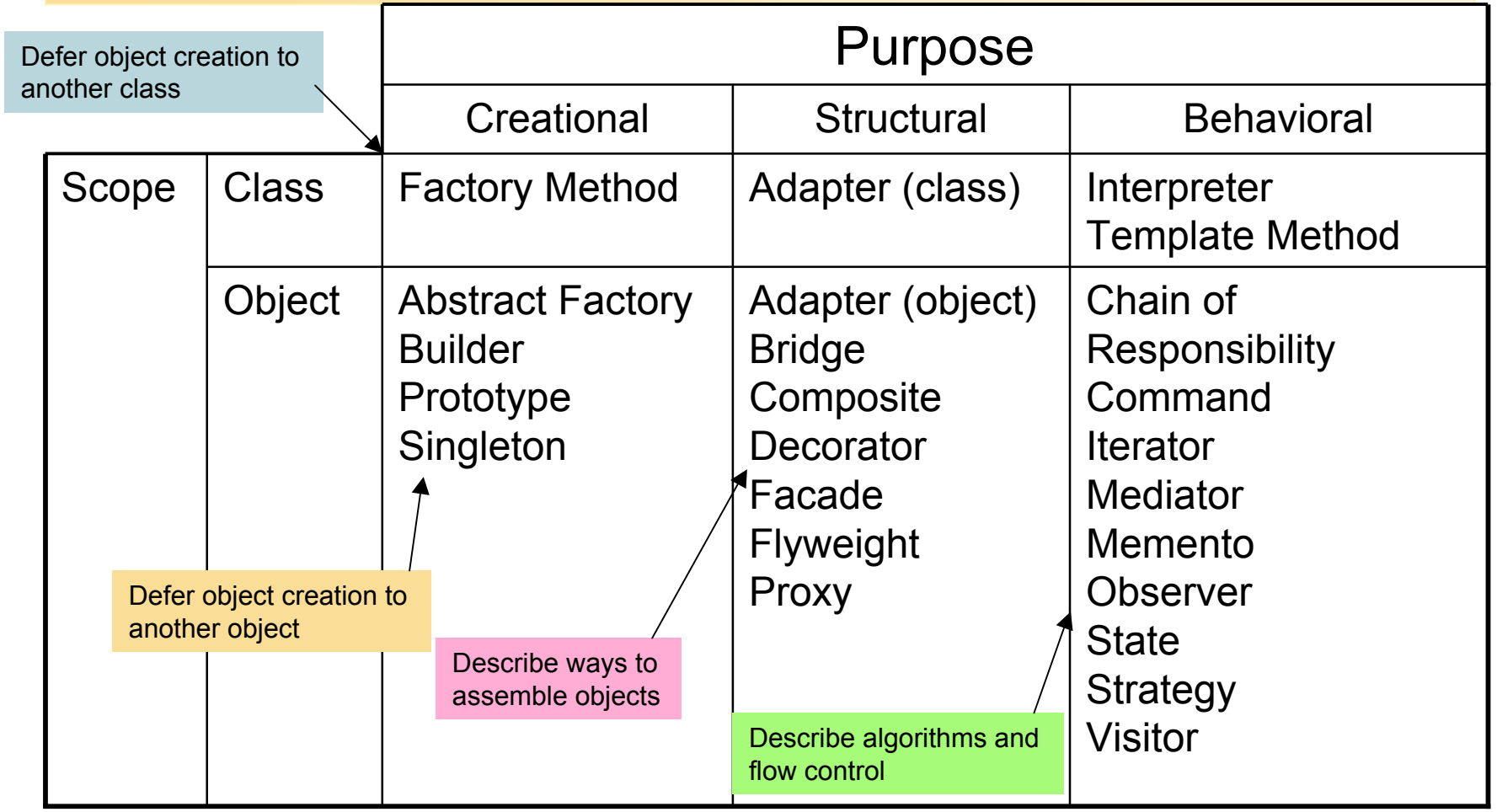


- Model
  - Implements algorithms (business logic)
  - Independent of environment
- View:
  - Communicates with environment
  - Implements I/O interface for model
- Controller:
  - Controls data exchange (notification protocol) between model and view

Model/View/Controller (contd.)

- MVC decouples views from models – more general:
  - Decoupling objects so that changes to one can affect any number of others
  - without requiring the object to know details of the others
  - **Observer pattern** solves the more general problem
- MVC allows view to be nested:
  - CompositeView objects act just as View objects
  - **Composite pattern** describes the more general problem of grouping primitive and composite objects into new objects with identical interfaces
- MVC controls appearance of view by controller:
  - Example of the more general **Strategy pattern**
- MVC uses **Factory** and **Decorator patterns** as well

Design Pattern Space



How Design Patterns Solve Design Problems

- Finding Appropriate Objects
  - Decomposing a system into objects is the hard part
  - OO-designs often end up with classes with no counterparts in real world (low-level classes like arrays)
  - Strict modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrows
  - Design patterns identify less-obvious abstractions
- Determining Object Granularity
  - Objects can vary tremendously in size and number
  - **Facade pattern** describes how to represent subsystems as objects
  - **Flyweight pattern** describes how to support huge numbers of objects

Specifying Object Interfaces

- Interface:
  - Set of all signatures defined by an object's operations
  - Any request matching a signature in the objects interface may be sent to the object
  - Interfaces may contain other interfaces as subsets
- Type:
  - Denotes a particular interfaces
  - An object may have many types
  - Widely different object may share a type
  - Objects of the same type need only share parts of their interfaces
  - A **subtype** contains the interface of its **supertype**
- Dynamic binding, polymorphism

Program to an interface, not an implementation

- Manipulate objects solely in terms of interfaces defined by abstract classes!

- Benefits:
  1. Clients remain unaware of the specific types of objects they use.
  2. Clients remain unaware of the classes that implement the objects. Clients only know about abstract class(es) defining the interfaces

  - Do not declare variables to be instances of particular concrete classes
  - Use creational patterns to create actual objects.

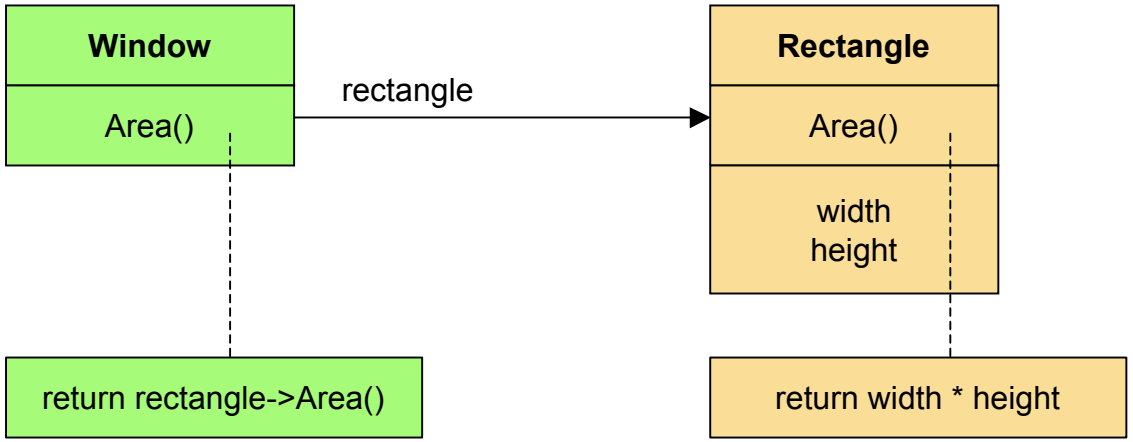
Favor object composition over class inheritance

- **White-box** reuse:
  - Reuse by subclassing (class inheritance)
  - Internals of parent classes are often visible to subclasses
  - works statically, compile-time approach
  - Inheritance breaks encapsulation
- **Black-box** reuse:
  - Reuse by object composition
  - Requires objects to have well-defined interfaces
  - No internal details of objects are visible

Delegation

Makes composition as powerful for reuse as inheritance

- Two objects involved in handling requests
- Explicit object references, no this-pointer
- Extreme example of object composition to achieve code reuse



But: Dynamic, hard to understand, run-time inefficiencies

Designing for Change – Causes for Redesign (I)

- Creating an object by specifying a class explicitly
  - Commits to a particular implementation instead of an interface
  - Can complicate future changes
  - Create objects indirectly
  - Patterns: Abstract Factory, Factory Method, Prototype
- Dependence on specific operations
  - Commits to one way of satisfying a request
  - Compile-time and runtime modifications to request handling can be simplified by avoiding hard-coded requests
  - Patterns: Chain of Responsibility, Command

Causes for Redesign (II)

- Dependence on hardware and software platform
  - External OS-APIs vary
  - Design system to limit platform dependencies
  - Patterns: Abstract Factory, Bridge
- Dependence on object representations or implementations
  - Clients that know how an object is represented, stored, located, or implemented might need to be changed when object changes
  - Hide information from clients to avoid cascading changes
  - Patterns: Abstract factory, Bridge, Memento, Proxy

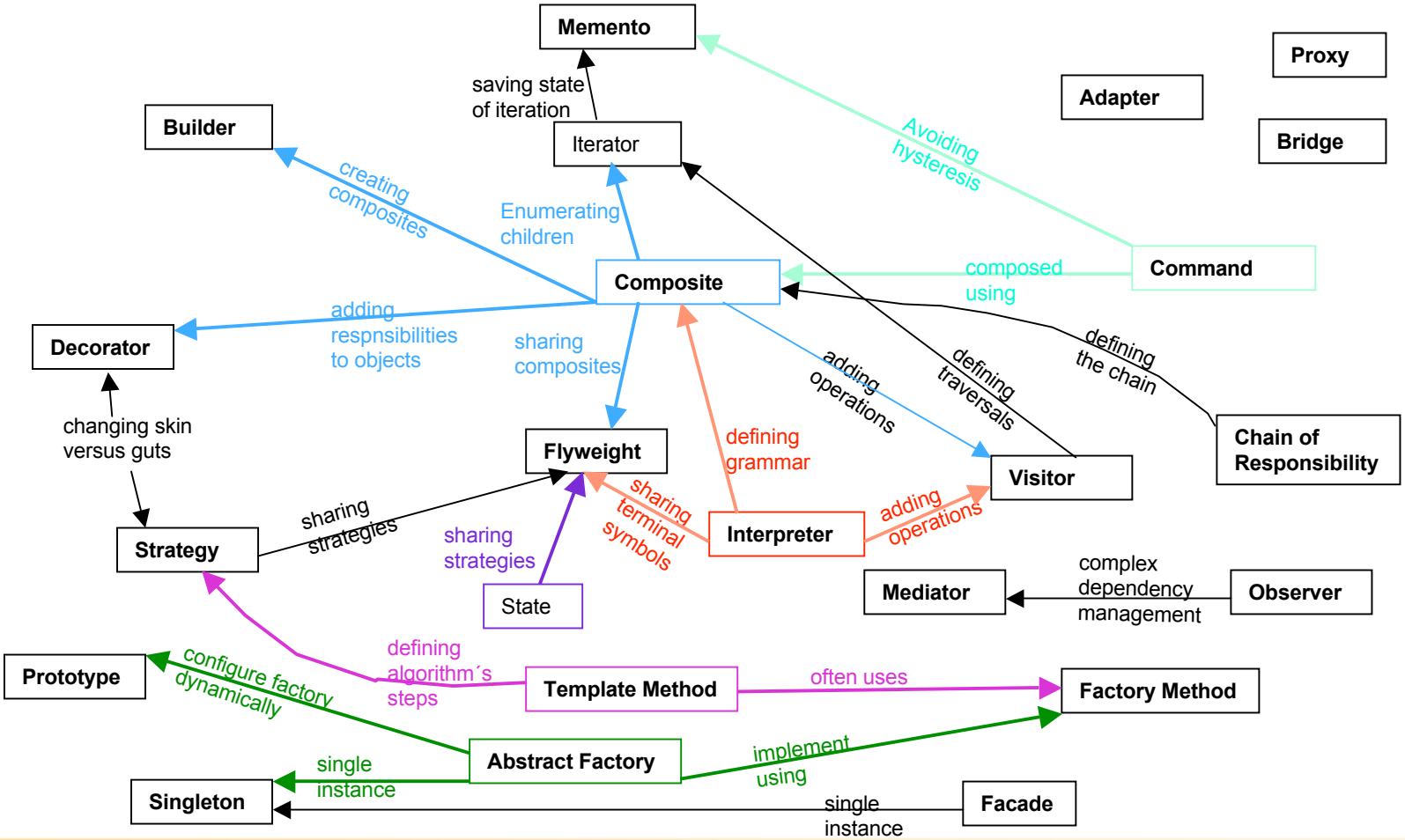
Causes for Redesign (III)

- Algorithmic dependencies
  - Algorithms are often extended, optimized, and replaced during development and reuses
  - Algorithms that are likely to change should be isolated
  - Patterns: Builder, Iterator, Strategy, Template Method, Visitor
- Tight coupling
  - Leads to monolithic systems
  - Tightly coupled classes are hard to reuse in isolation
  - Patterns: Abstract Factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer

Causes for Redesign (IV)

- Extending functionality by subclassing
  - Requires in-depth understanding of the parent class
  - Overriding one operation might require overriding another
  - Can lead to an explosion of classes (for simple extensions)
  - Patterns: Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy
- Inability to alter classes conveniently
  - Sources not available
  - Change might require modifying lots of existing classes
  - Patterns: Adapter, Decorator, Visitor

Relations among Design Patterns



List of Design Patterns

- Creational Patterns
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton
- Structural Patterns
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy
- Behavioral Patterns
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor

How to Select a Design Pattern

- Consider how design patterns solve design problems
  - Find appropriate objects
  - Determine object granularity
  - Specifiy object interfaces
- Scan intent sections
- Study how patterns interrelate
- Study patterns of like purpose
  - Creational, structural, behavioral patterns
- Examine cause of redesign
- Consider what should be variable in your design

How to Use a Design Pattern

- Read the pattern once through for an overview
  - Study applicability and consequences
- Study Structure, Participants, Collaborations
- Choose names for pattern participants that are meaningful in the application context
- Define the classes
  - Declare interfaces, inheritance relationships; define instance variables
  - Identify existing classes in your app that the pattern will affect
- Define application-specific names for ops in the pattern
- Implement operations to carry out responsibilities and collaborations in the pattern

Design aspects that design patterns let you vary

Purpose	Design Pattern	Aspect(s) that can vary
Creational	Abstract Factory	Families of product objects
	Builder	How a composite object gets created
	Factory Method	Sublass of object that is instantiated
	Prototype	Class of object that is instantiated
	Singleton	The sole instance of a class
Structural	Adapter	Interface to an object
	Bridge	Implementation of an object
	Composite	Structure and composition of an object
	Decorator	Responsibilities of an object without subclassing
	Facade	Interface to a subsystem
	Flyweight	Storage cost of objects
	Proxy	How an object is accessed, its location

Design aspects that design patterns let you vary (contd.)

Purpose	Design Pattern	Aspect(s) that can vary
Behavioral	Chain of Resp.	Object that can fulfill a request
	Command	When and how a request is fulfilled
	Interpreter	Grammar and interpretation of a language
	Iterator	How an aggregate‘s elements are accessed, traversed
	Mediator	How and which objects interact with each other
	Memento	What private information is stored outside an object, and when
	Observer	Number of objects that depend on another object; how the dependent objects stay up to date
	Stae	States of an object
	Strategy	An algorithm
	Template Method	Steps of an algorithm
	Visitor	Operations that can be applied to object(s) without changing their class(es)

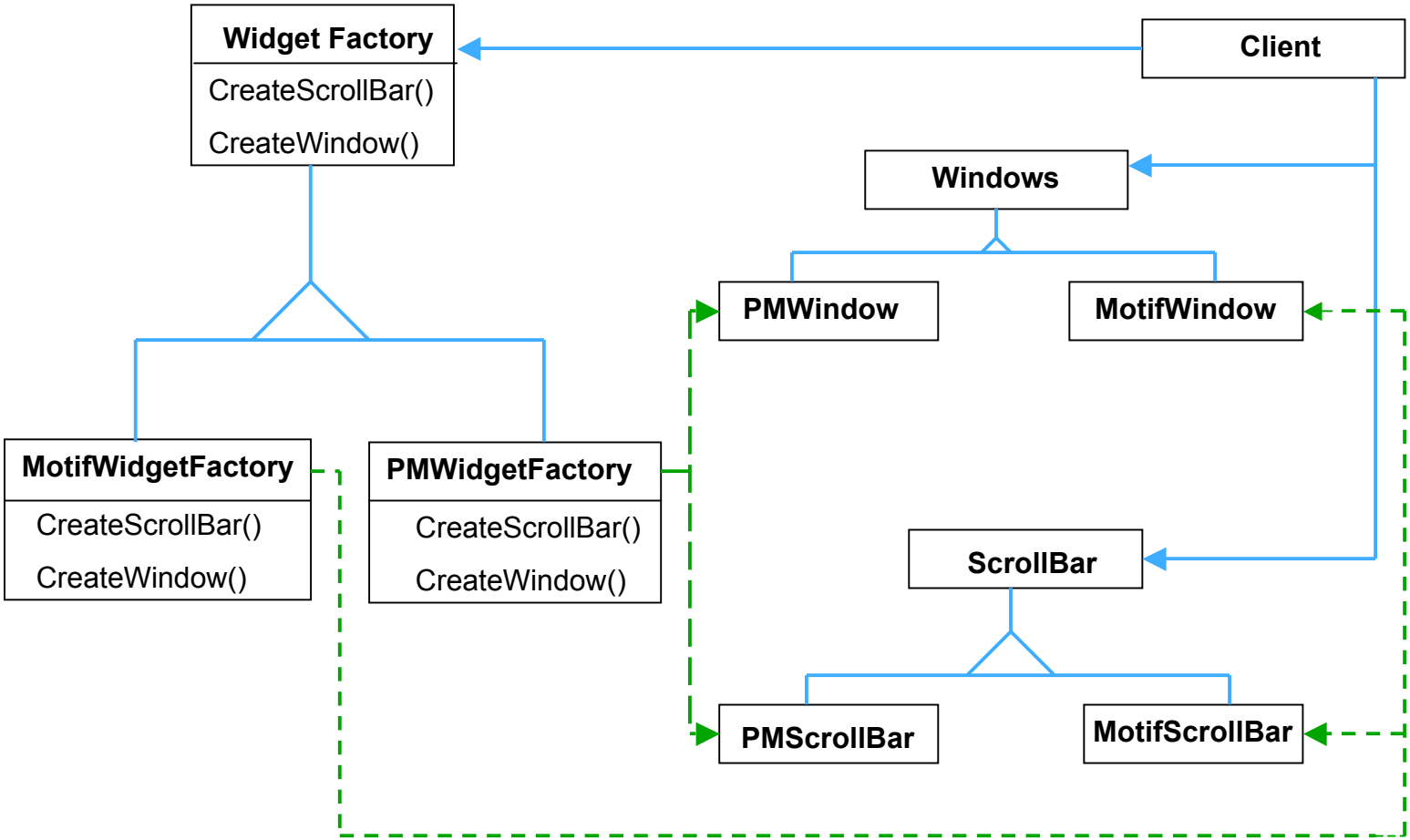
Creational Patterns

- Abstract the instantiation process
  - Make a system independent of how ist objects are created, composed, and represented
- Important if systems evolve to depend more on object composition than on class inheritance
  - Emphasis shifts from hardcoding fixed sets of behaviors towards a smaller set of composable fundamental behaviors
- Encapsulate knowledge about concrete classes a system uses
- Hide how instances of classes are created and put together

ABSTRACT FACTORY (Object Creational)

- Intent:
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes
  - Also known as: Kit
- Motivation:
  - User interface toolkit supports multiple look-and-feel standards (Motif, Presentation Manager)
  - Different appearances and behaviors for UI widgets
  - Apps should not hard-code its widgets
- Solution:
  - Abstract WidgetFactory class
  - Interfaces for creating each basic kind of widget
  - Abstract class for each kind of widgets,
  - Concrete classes implement specific look-and-feel

ABSTRACT FACTORY
Motivation

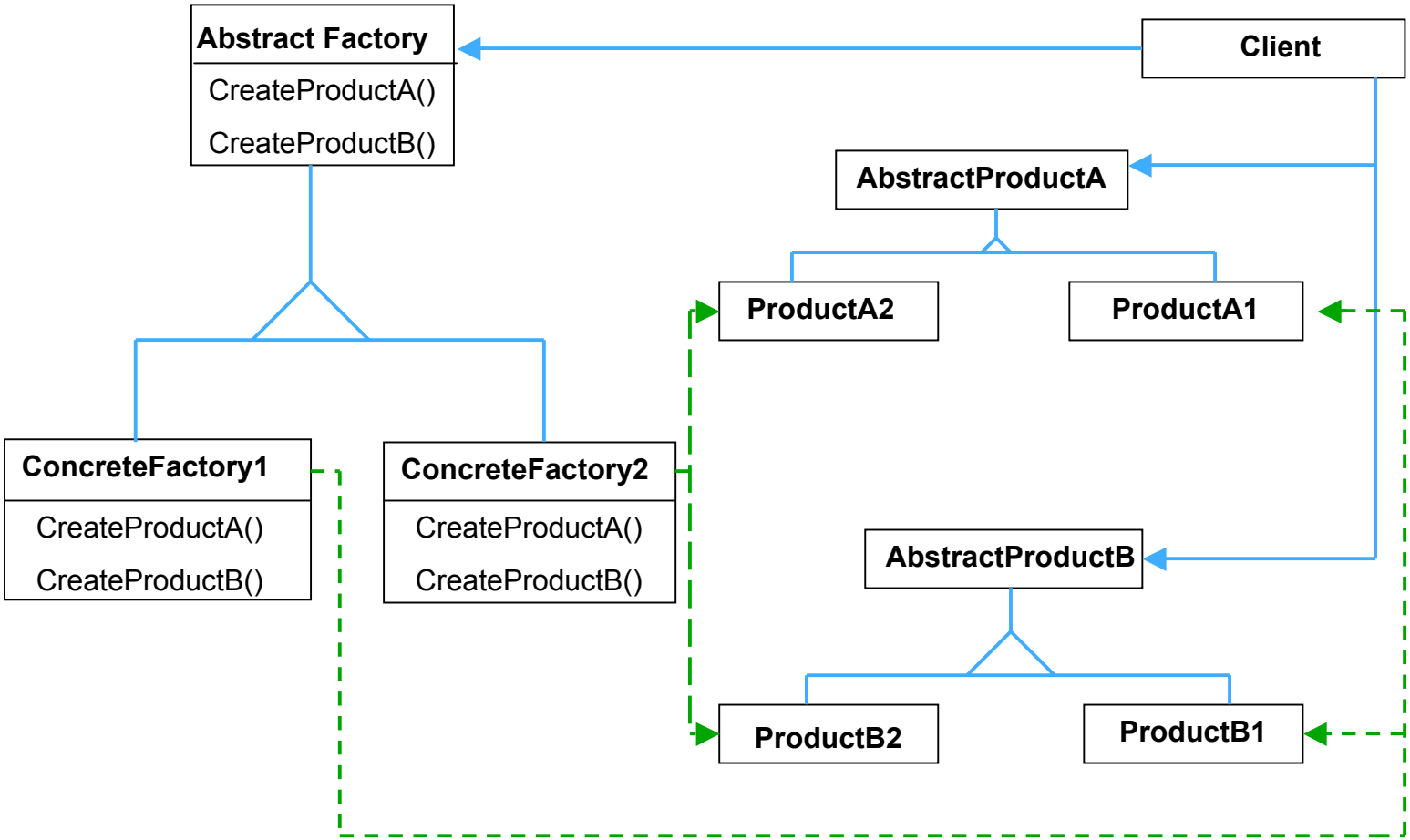


Applicability

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 produces
- 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



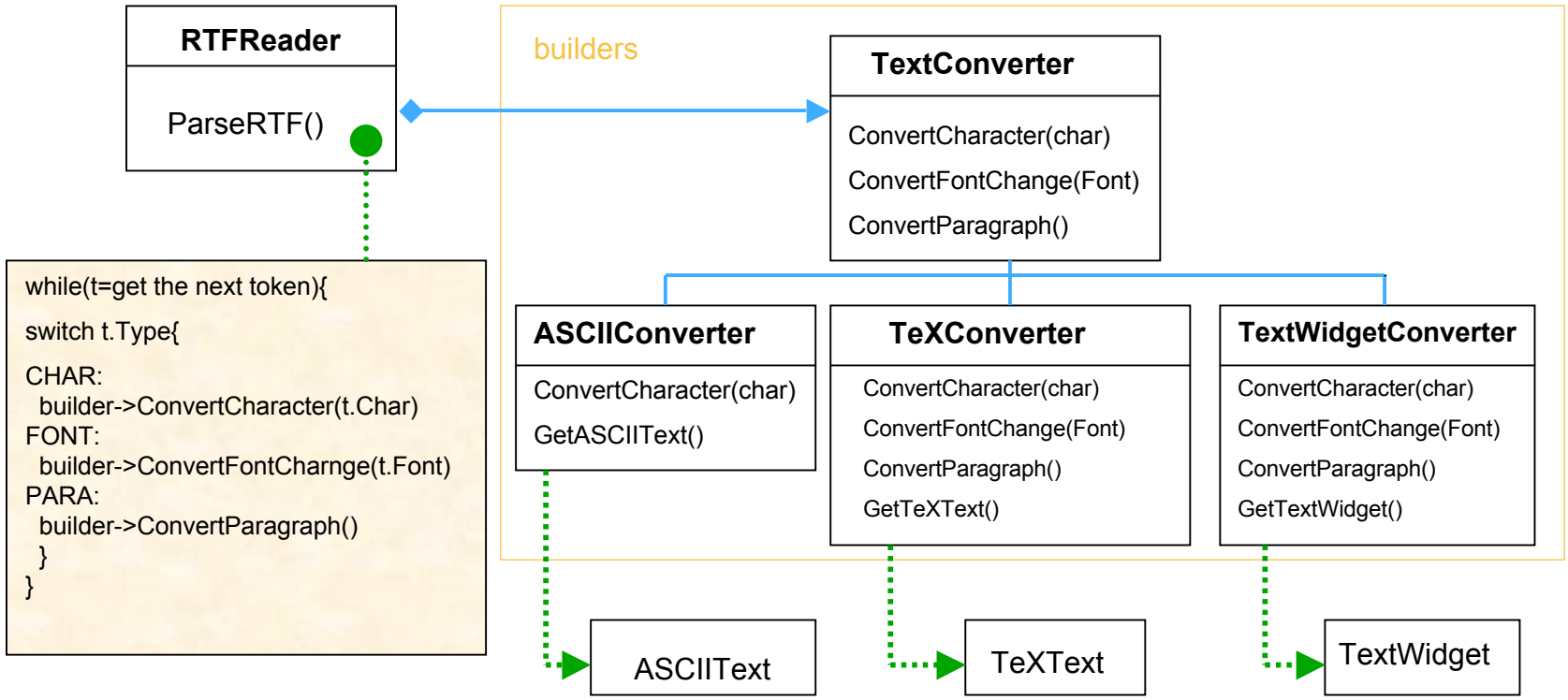
ABSTRACT FACTORY
Participants

- AbstractFactory
  - Declares interface for operations that create abstract product objects
- ConcreteFactory
  - Implements 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 concrete factory
  - Implements the abstract product interface
- Client
  - Uses only interfaces declared by AbstractFactory and AbstractProduct classes

BUILDER
(Object Creational)

- Intent:
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations
- Motivation:
  - RTF reader should be able to convert RTF to many text format
  - Adding new conversions without modifying the reader should be easy
- Solution:
  - Configure RTFReader class with a TextConverter object
  - Subclasses of TextConverter specialize in different conversions and formats
  - TextWidgetConverter will produce a complex UI object and lets the user see and edit the text

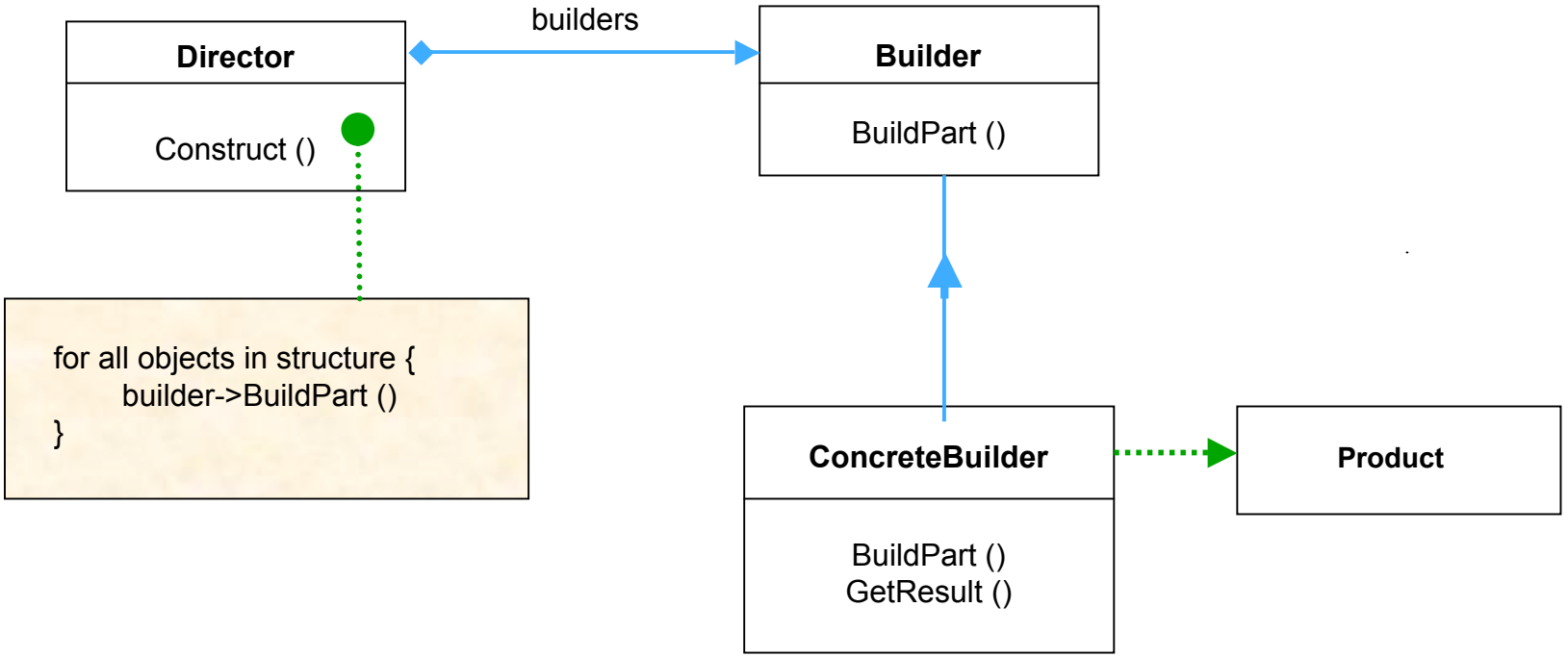
BUILDER
Motivation



Applicability

- Use the Builder pattern when
  - The algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled
  - The construction process must allow different representations for the object that is constructed

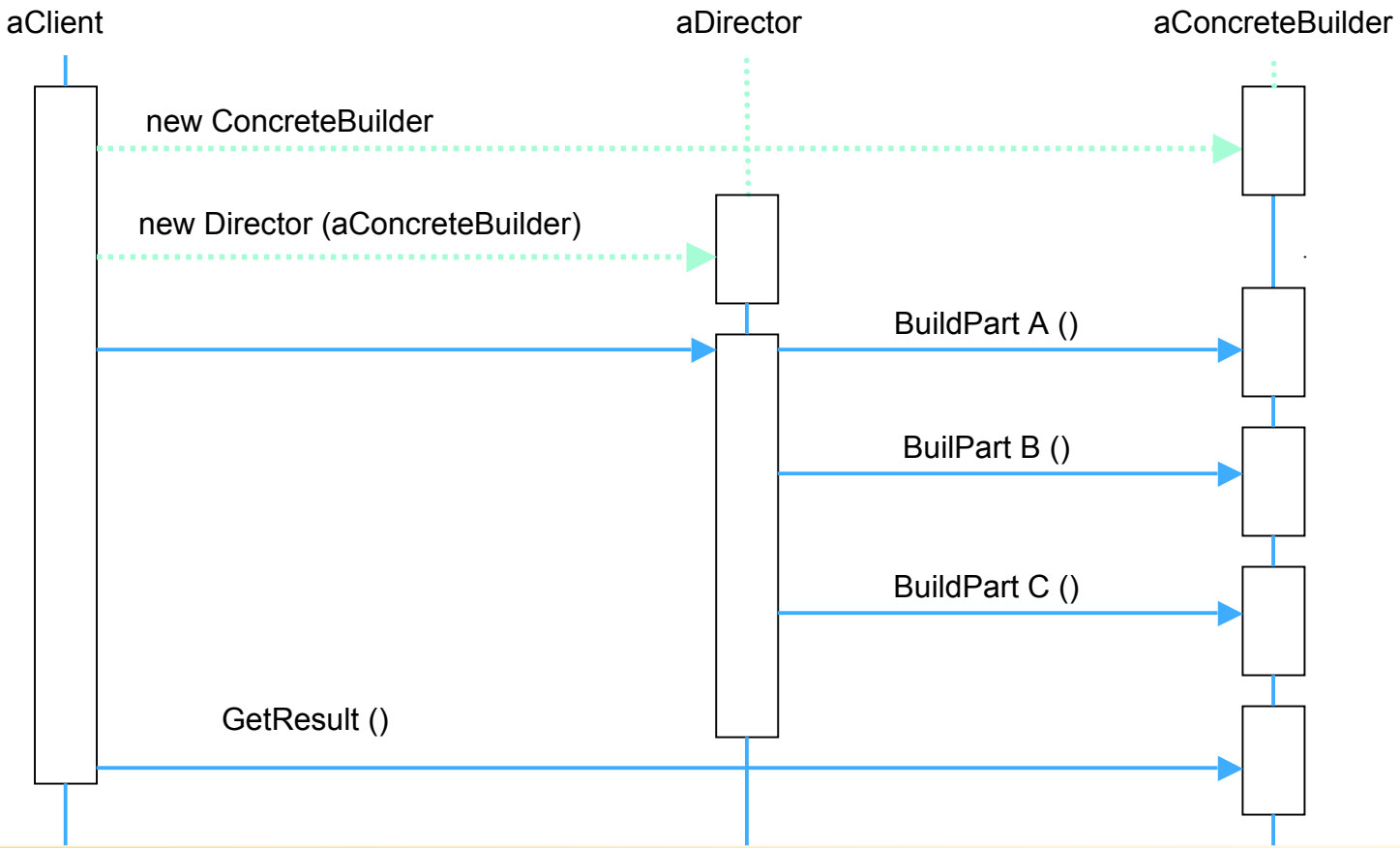
BUILDER
Structure



Builder - Collaborations

- Client creates Director object and configures it with the desired Builder object
- Director notifies Builder whenever a part of the product should be built
- Builder handles requests from the Director and adds parts to the product
- Client retrieves the product from the Builder

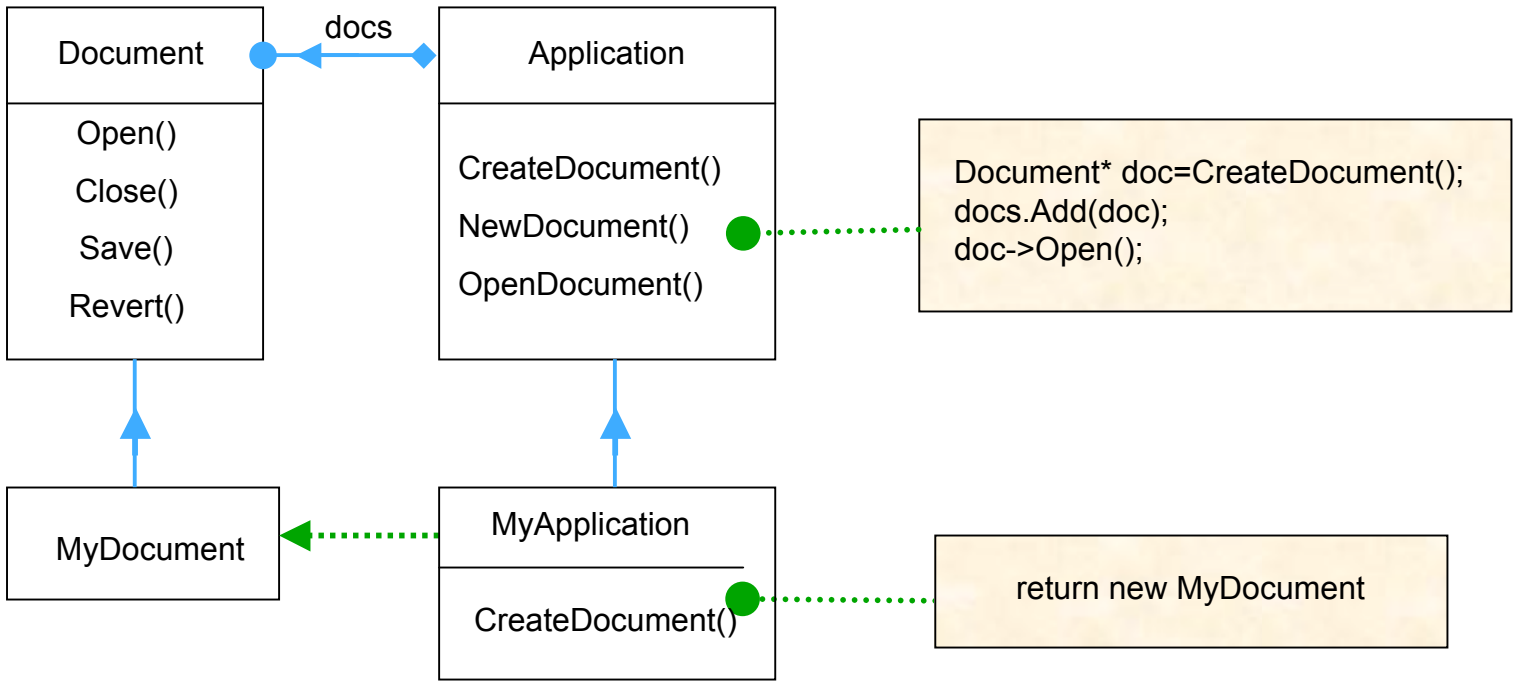
BUILDER
Collaborations



FACTORY METHOD
(Class Creational)

- Intent:
  - Define an interface for creating an object, but let subclasses decide which class to instantiate.
  - Factory Method lets a class defer instantiation to subclasses.
  - Also known as: Virtual Constructor
- Motivation:
  - Framework use abstract classes to define and maintain relationships between objects
  - Framework has to create objects as well - must instantiate classes but only knows about abstract classes - which it cannot instantiate
  - Factory method encapsulates knowledge of which subclass to create - moves this knowledge out of the framework

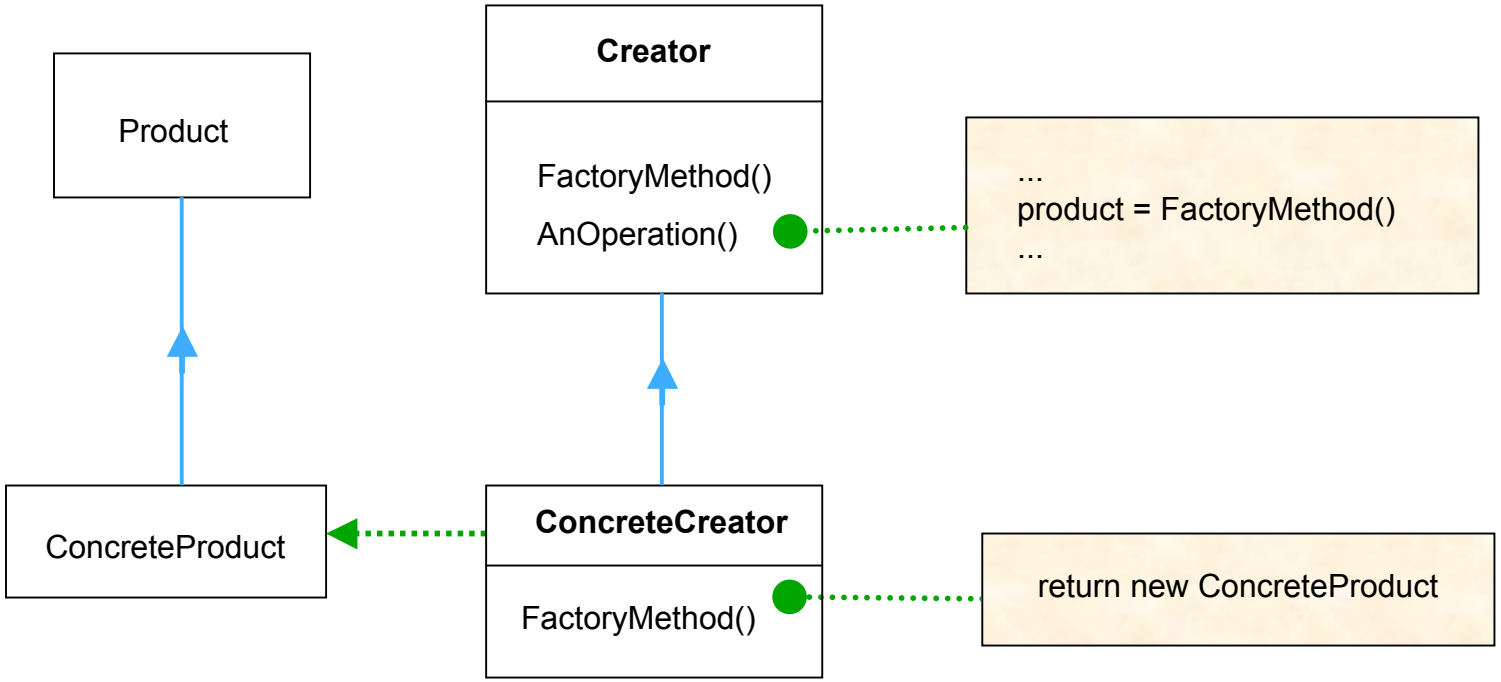
FACTORY METHOD
Motivation



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



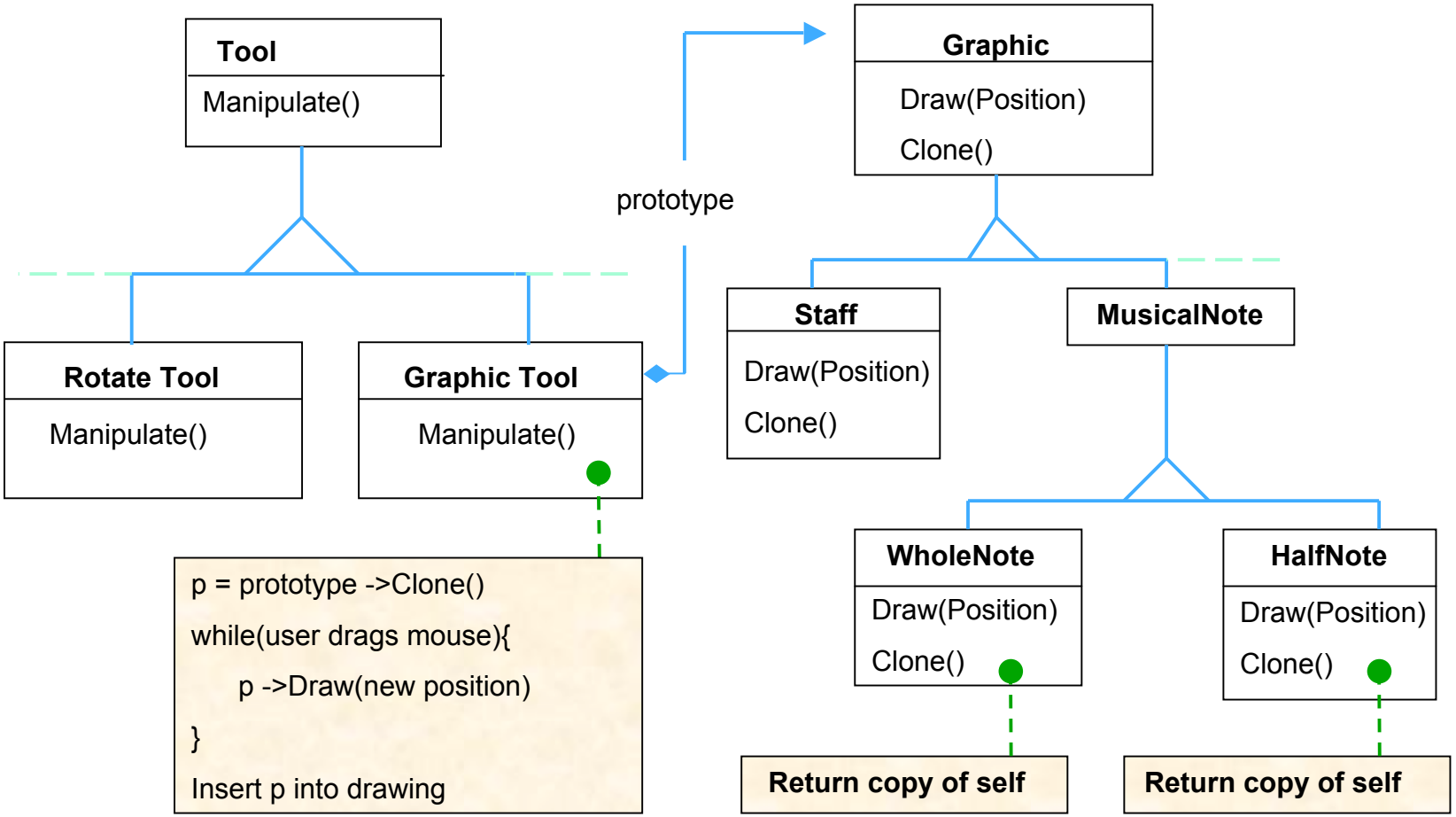
Participants

- Product
  - Defines the interface of objects the factory method creates
- ConcreteProduct
  - Implements the product interface
- Creator
  - Declares the factory method which returns object of type product
  - May contain a default implementation of the factory method
  - Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate Concrete Product.
- ConcreteCreator
  - Overrides factory method to return instance of ConcreteProduct

PROTOTYPE
(Object Creational)

- Intent:
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Motivation:
  - Framework implements Graphic class for graphical components and GraphicTool class for tools manipulating/creating those components
  - Actual graphical components are application-specific
  - How to parameterize instances of GraphicTool class with type of objects to create?
  - Solution: create new objects in GraphicTool by cloning a **prototype** object instance

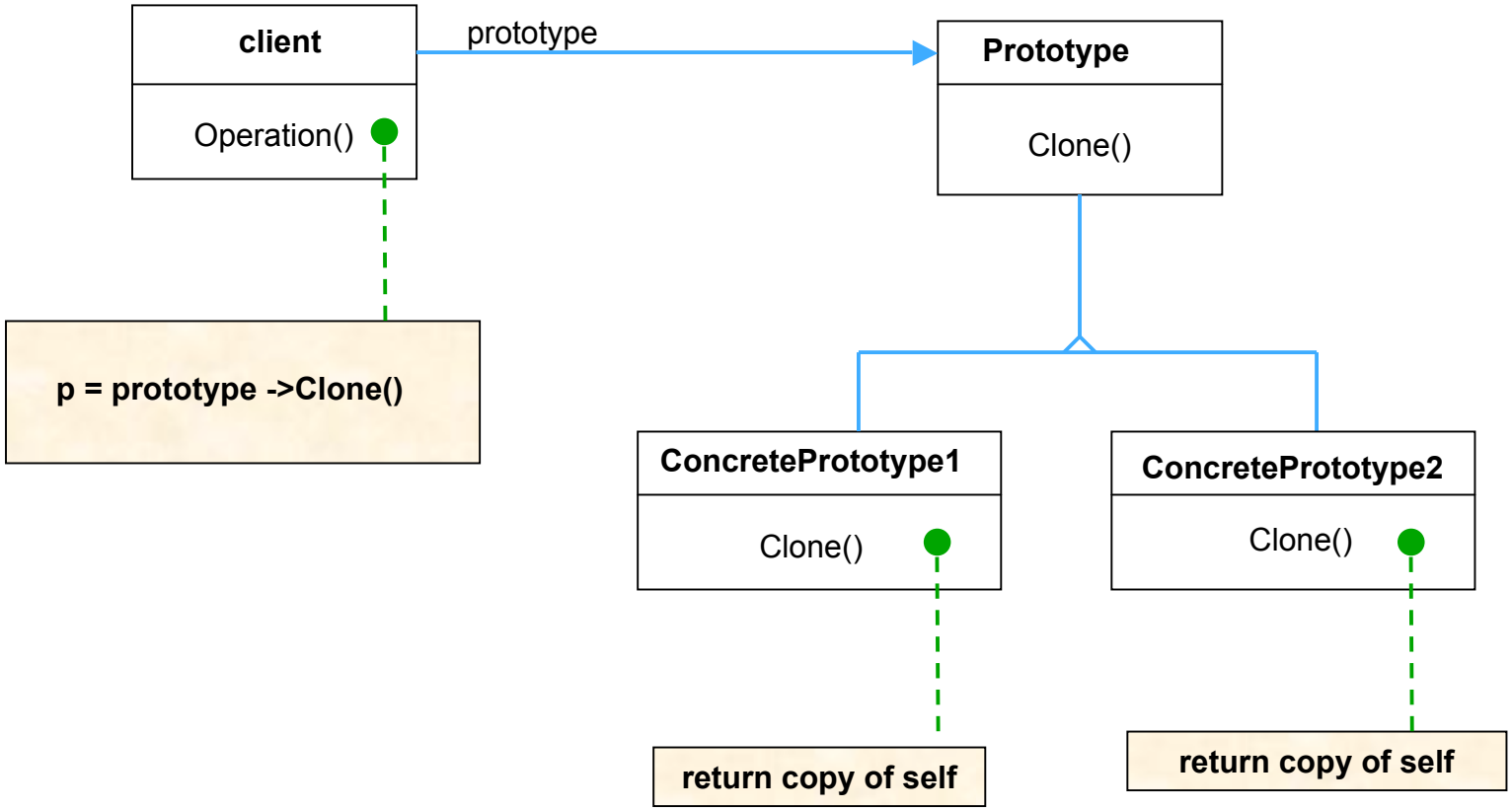
PROTOTYPE
Motivation



Applicability

- Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented;
  - 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 state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

PROTOTYPE
Structure



Participants and Collaborations

Participants:

- Prototype (Graphic)
  - Declares an interface for cloning itself
- ConcretePrototype (Staff, WholeNote, HalfNote)
  - Implements an interface for cloning itself
- Client (GraphicTool)
  - Creates a new object by asking a prototype to clone itself

Collaborations:

- A client asks a prototype to clone Itself.

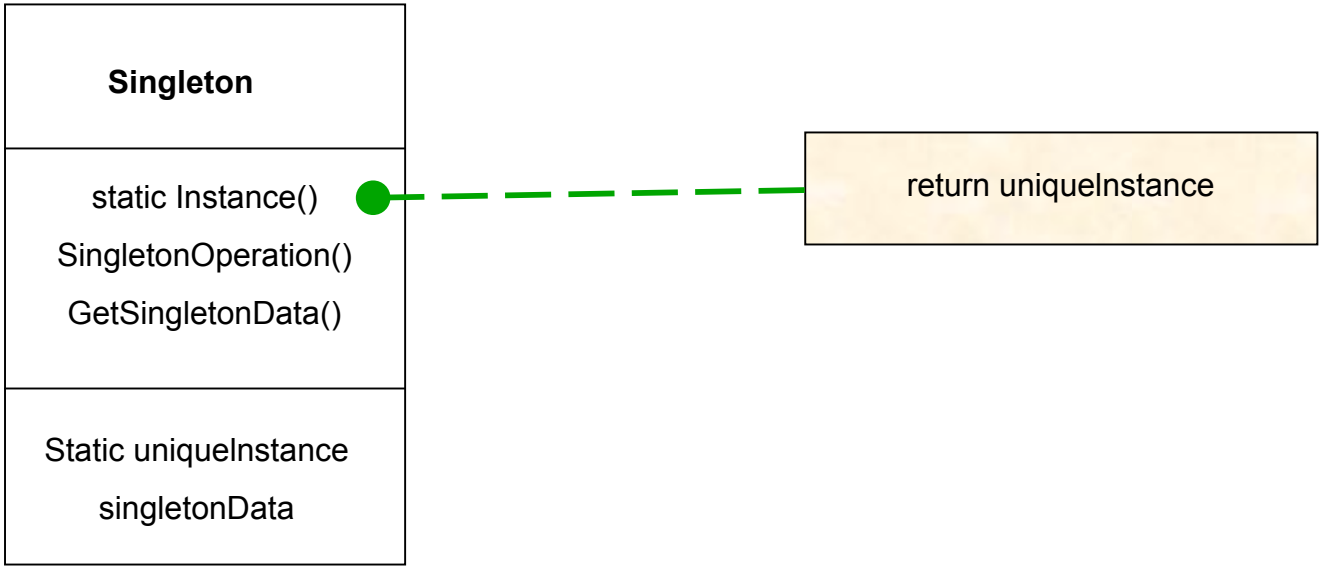
SINGELTON
(Object Creational)

- Intent:
  - Ensure a class only has one instance, and provide a global point of access to it.
- Motivation:
  - Some classes should have exactly one instance (one print spooler, one file system, one window manager)
  - A global variable makes an object accessible but doesn't prohibit instantiation of multiple objects
  - Class should be responsible for keeping track of its sole interface

Applicability

- Use the Singleton pattern when
  - there must be exactly one instance of a class, and it must be accessible to clients 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
Structure



Participants and Collaborations

- Singleton:
- Defines an instance operation that lets clients access its unique interface
- Instance is a class operation (static in Java)
- May be responsible for creating its own unique instance
- Collaborations:
- Clients access a Singleton instance solely through Singleton's Instance operation.