



Iterator pattern

Acknowledgement: Eric Braude

Let's try this...

Data structures:

- Array
- Binary Tree
- Vector
- Linked list
- Hash table

Algorithm:

- Sort
- Find
- Merge

How many permutations to develop/maintain ?



Iterators – The Problem

○ Given

- An aggregate collection of objects

○ Desired

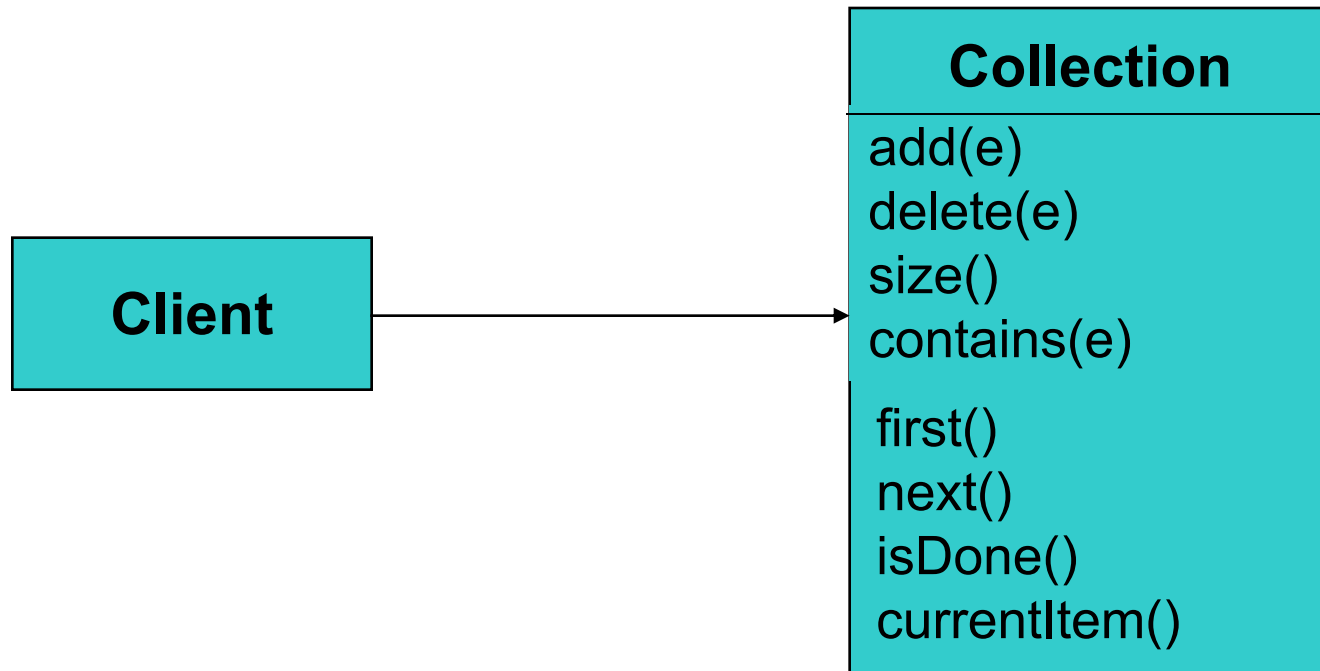
- Access all elements in the collection
- Do not expose internal structure
- Access is independent of the collection
- Multiple accesses can be done independent of each other
- Collection can be maintained independent of the access

○ Solution

- Iterator object controls access

Embedded Iterator

- Embed Iterator support in the Collection





Issues with Embedded Iterators

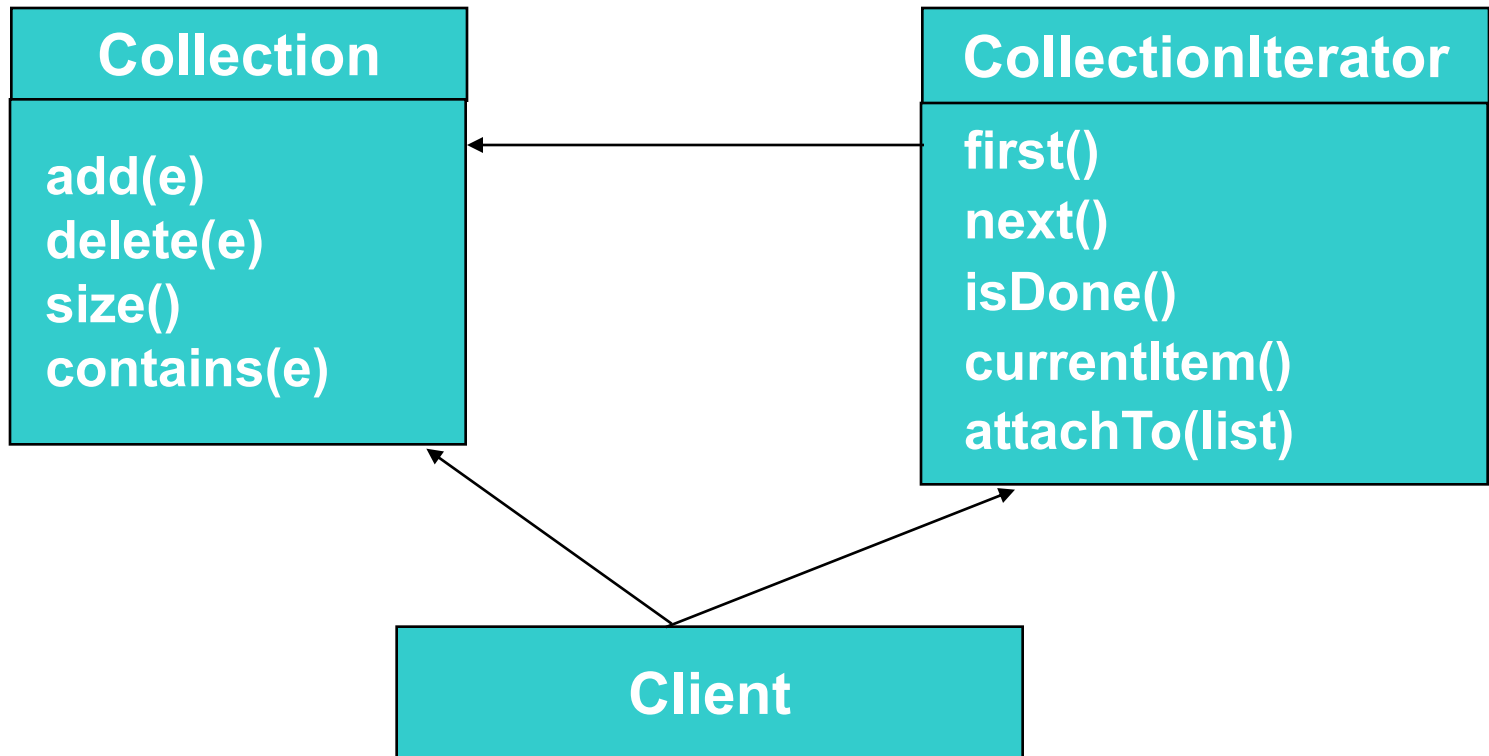
- Mixes structure maintenance and traversal
- Only one traversal at a time
- Adds to collection interface
- Adds to collection implementation
- Clumsy when adding different traversal algorithms:
 - Lists: forward / reverse traversal
 - Trees: pre / post / inorder
 - Graphs: depth first vs. breadth first



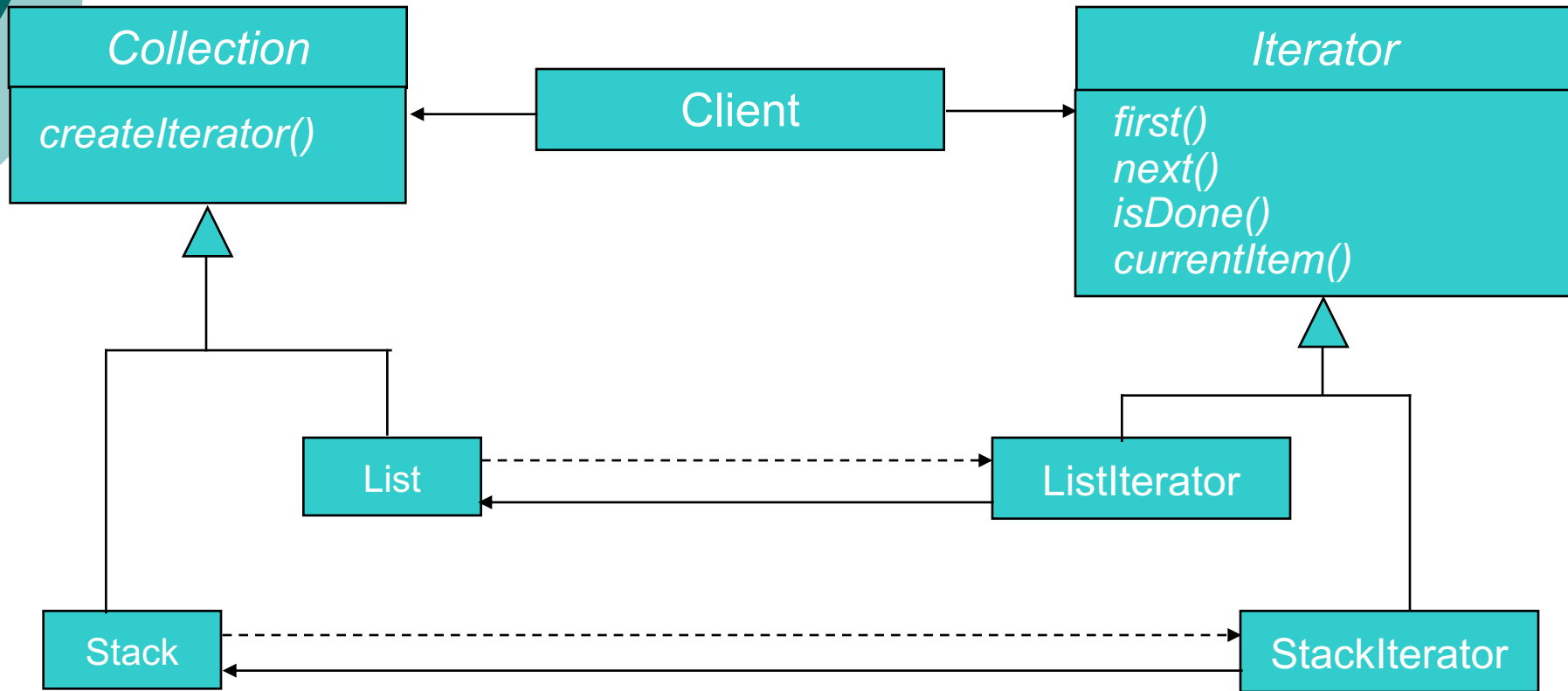
Alternative: Separate Iterator Class & Objects

- Create iterators as needed for a collection
- Attach to collection at creation time or via attach method
- Use iterator object, not collection, to access elements.

Separate Collection Iterator



Abstract Collections & Iterators





Consequences

- Supports various traversal algorithms via different iterators over same collection.
- Simplifies collection interface: At most must return different iterators
- More than one traversal can be in progress



Internal and External Iterators

- Examples so far are external iterators
 - Client object does iteration using provided methods
- Internal iterators
 - The iterator controls the traversal.
 - Client hands iterator some operation to perform on each element in aggregate.
 - Iterator traverses aggregate (Iteration done by iterator object itself)



Builder pattern

The Problem

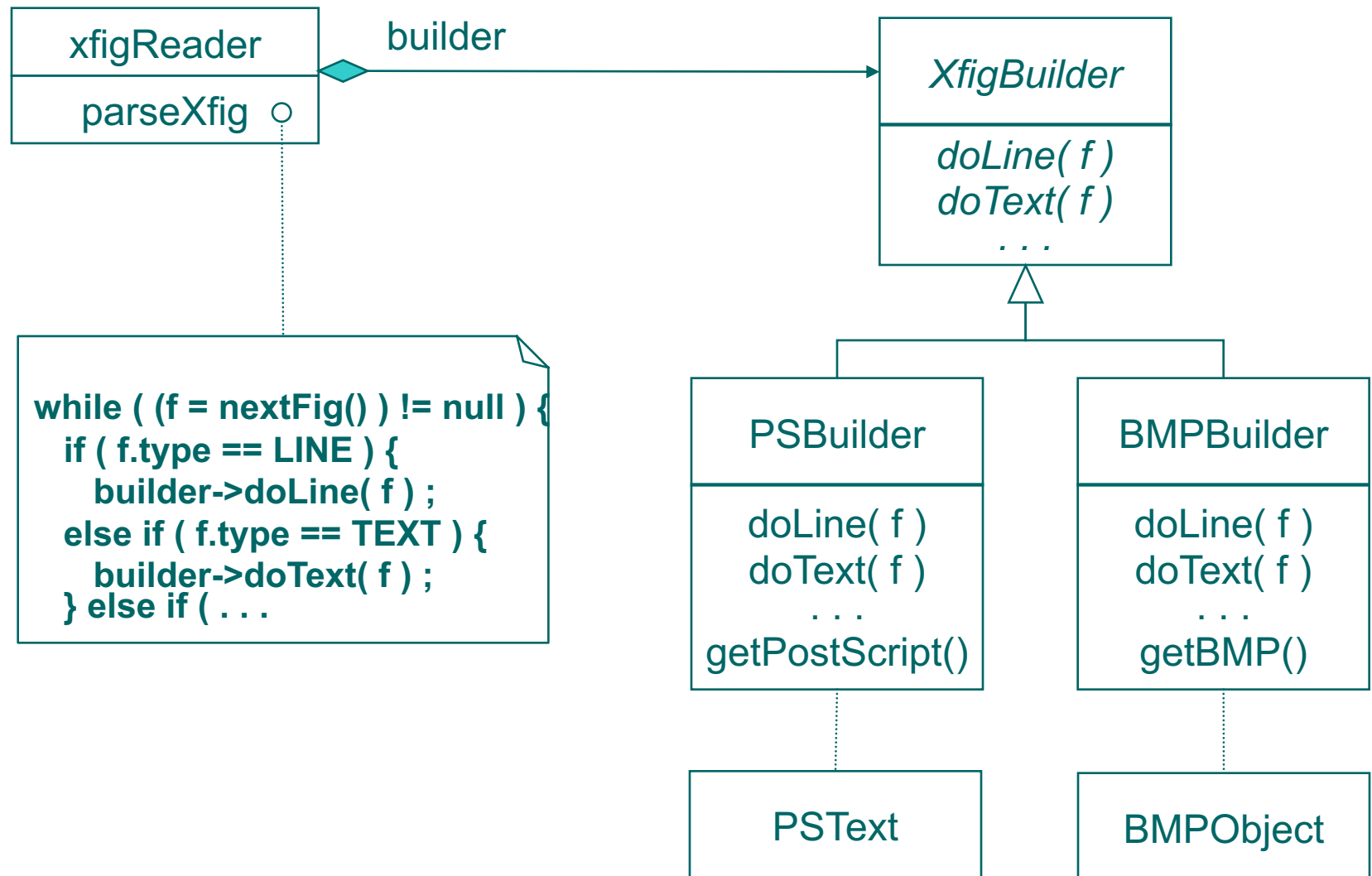
- Need to construct complex object incrementally.
- There may be many variants of complex object
- Example: *fig2dev*
 - One input -- the *xfig* file of figure descriptions
 - Many output objects:
 - Postscript
 - BMP for Windows
 - TIFF for fax
- **fig2dev -L** *language* [*fig-file* [*out-file*]]
- How to abstract the construction process?



Possible Solution: Builder

- Builder provides abstract interface for incremental construction
- Concrete builders provide variant on this interface
- Select desired concrete builder and provide to client
- Client simply invokes interface methods for each constituent "piece"
- Parameterizes the construction of many variants.

A Builder for Xfig Format

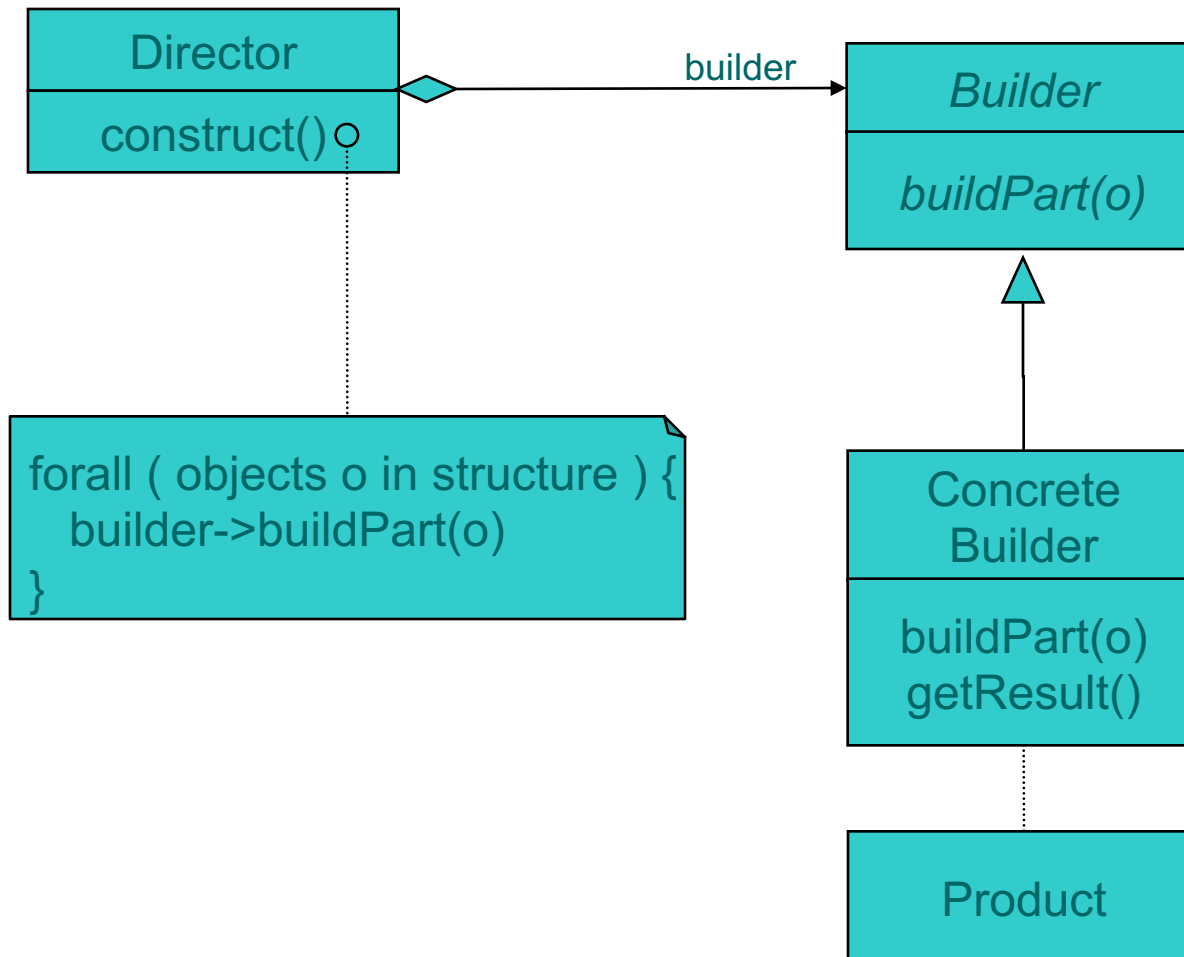




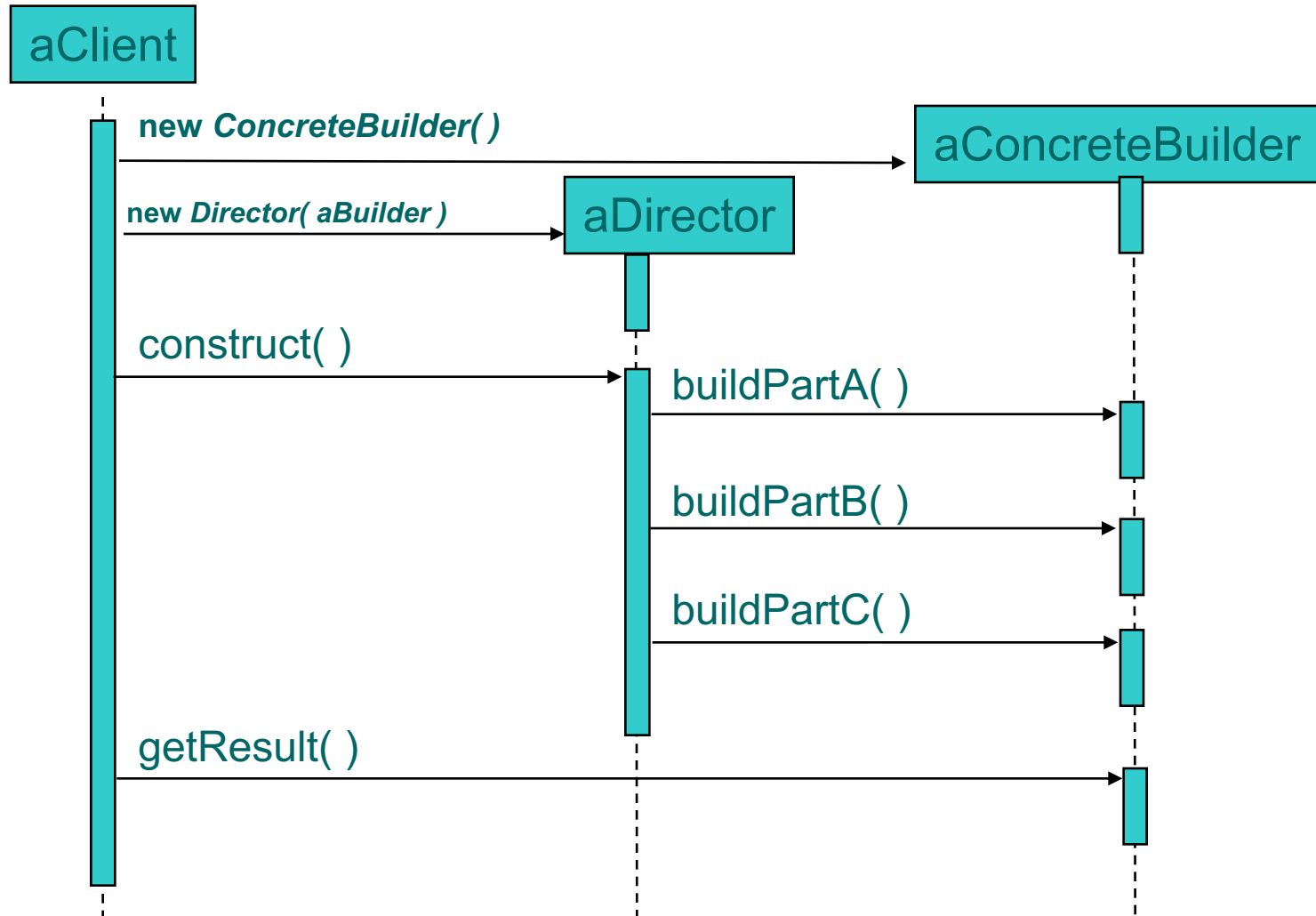
Use the Builder Pattern When

- Need algorithm / assembly independence
 - Algorithm for construction in the client
 - Specific parts of complex object hidden in builder
 - Assembly performed in builder
- Many variant representations
 - Construction fixed
 - Construction incremental
 - Many possible construction variants

General Builder Structure



Builder Interaction Diagram



Collaborations

- *Client* creates *Director* and configures it with a specific *Builder*
- *Director* notifies *Builder* when a part is to be added to the *Product*
- *Builder* adds parts requested to the *Product*
- *Builder* returns *Product* to *Client* on request



Consequence: Can Vary Product

- Can add new types of products built according to common process
- In our example, can add drivers for other forms of xfig output.
- Specifics of how assembly proceeds are hidden, as long as every legal assembly sequence can be accommodated

Consequence: Isolates Construction Algorithm From Representations

- Encapsulates *process* of construction in *Director*
 - Might construct objects from different inputs
 - Example: fig2dev driven by another drawing program format
- Encapsulates *product* of construction in *Builder*
- Can vary each independently as long as *Builder* interface is stable.
- Note: There may be semantic constraints on Builder interface sequencing:
 - E.g. each "startComposite()" needs a matching "endComposite()"



Consequence: Fine-Grained Construction Control

- Does not construct full product object in one-shot
- Supports step-by-step construction: Like building a house
- Only retrieve the product when Director is finished with the construction
- Changes to *Directors*
 - Must recognize new types of parts
 - Must know how to invoke construction of part inside the Builder
- Changes to *Builders*
 - Need to add a new part construction method to interface
 - Need to update all subclasses (e.g., all concrete builders)



Command pattern



The Problem

○ Problem

- An application has multiple commands being issued via different mechanisms

○ Desire: Decouple invoking of the action from

- Knowledge of how to perform it
- Knowledge of the receiver of request

○ Solution

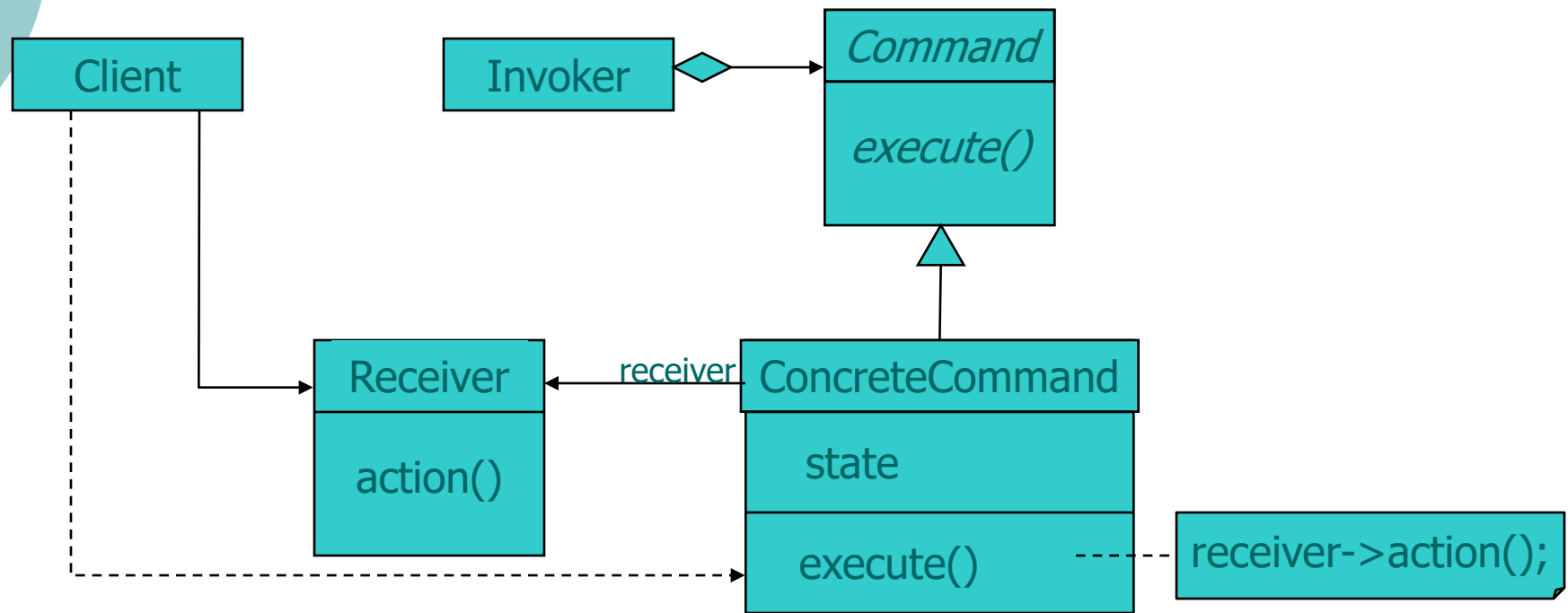
- Create **Command** objects that know how to execute the operations



Example

- Image manipulation program allows commands to be issued from
 - Toolbar item
 - Menu item
 - Script language
 - Fly-over menu
- All these interface elements should not need to know how to perform the operation

Pattern Structure

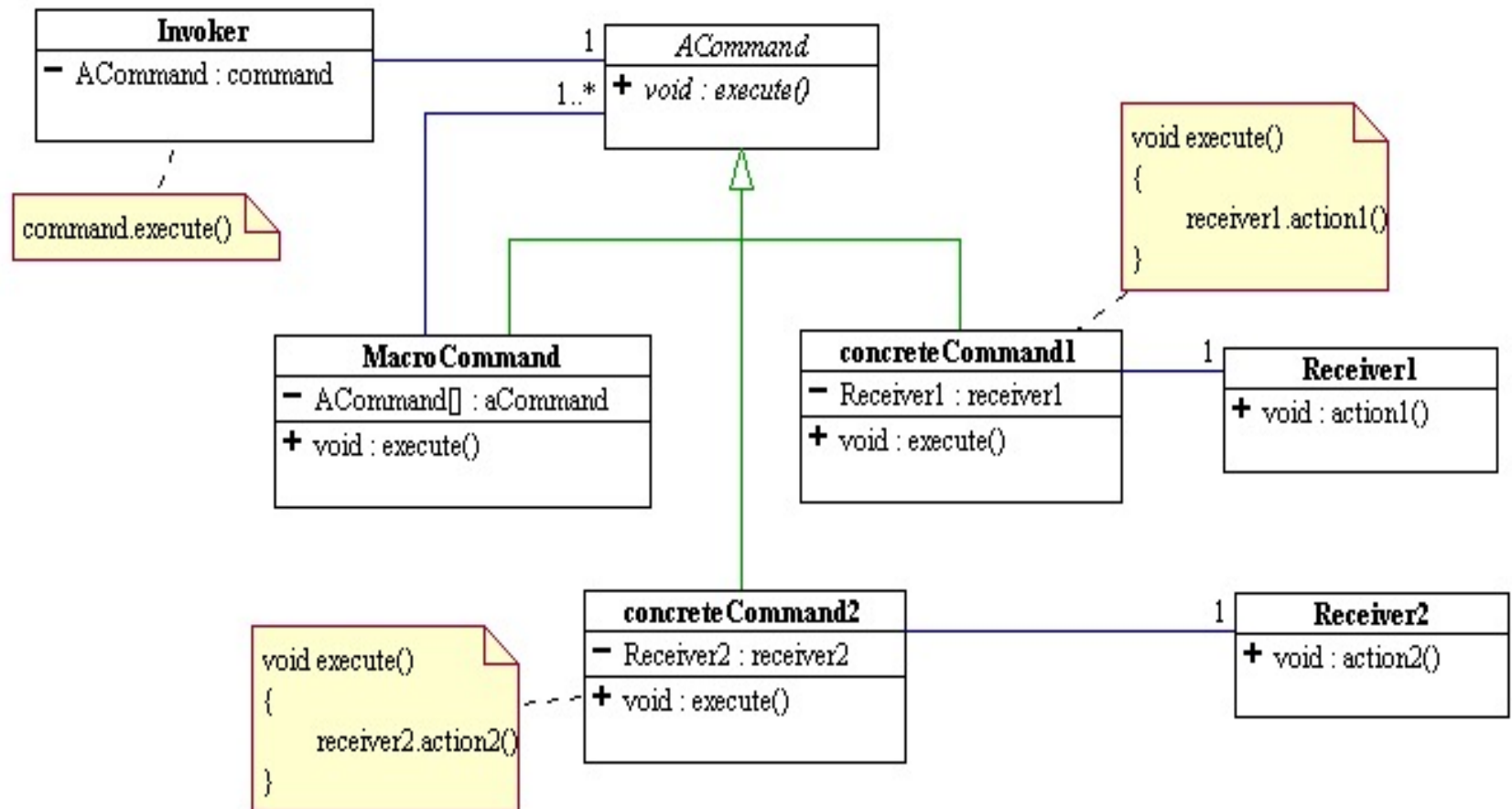




Participants

- **Command**
 - Interface for executing every operation
- **Concrete Command**
 - Implements operation
 - Binds receiver and action
- **Client**
 - Creates Concrete Command
 - Determines Receiver
- **Invoker**
 - Requests command to execute operation
- **Receiver**
 - Performs the operations needed

Example





Consequences

- Invocation is decoupled from execution
- Command is an object \Rightarrow it can be extended
- Macro commands accommodated
- Easy to add new commands to the application interface

Undo / Redo

- Unexecute operation
- How to return to original state of receiver?
 - Save state in receiver?
 - “Unaction” in receiver?
 - Save state in command? \Rightarrow **Memento**
- History list
 - Going backward
 - Going forward
 - Do you need a copy of command? \Rightarrow **Prototype**
- Reliable restoration over repeated undo/redo cycles