

Composite and Builder

Steven R. Bagley

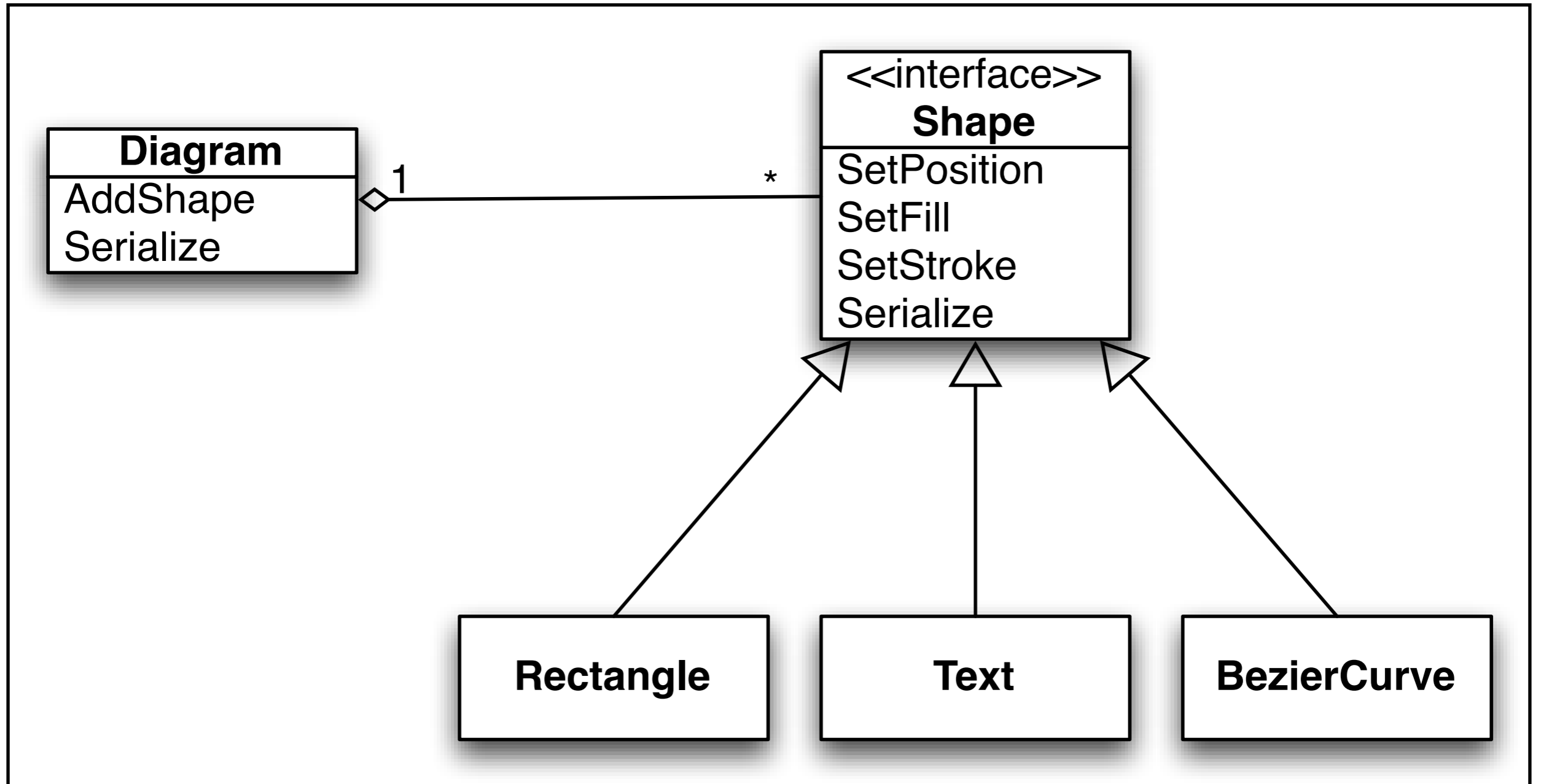
Object Creational Patterns

- Factory — Abstract creation
- Singleton — One and Only One instance
- Builder — Creating complex objects

Complex Objects

- Complex Object structures
- Data represented by many objects composed together
- Objects combined into a tree or graph structure recursively
- Provides programmatic interface for manipulating the data

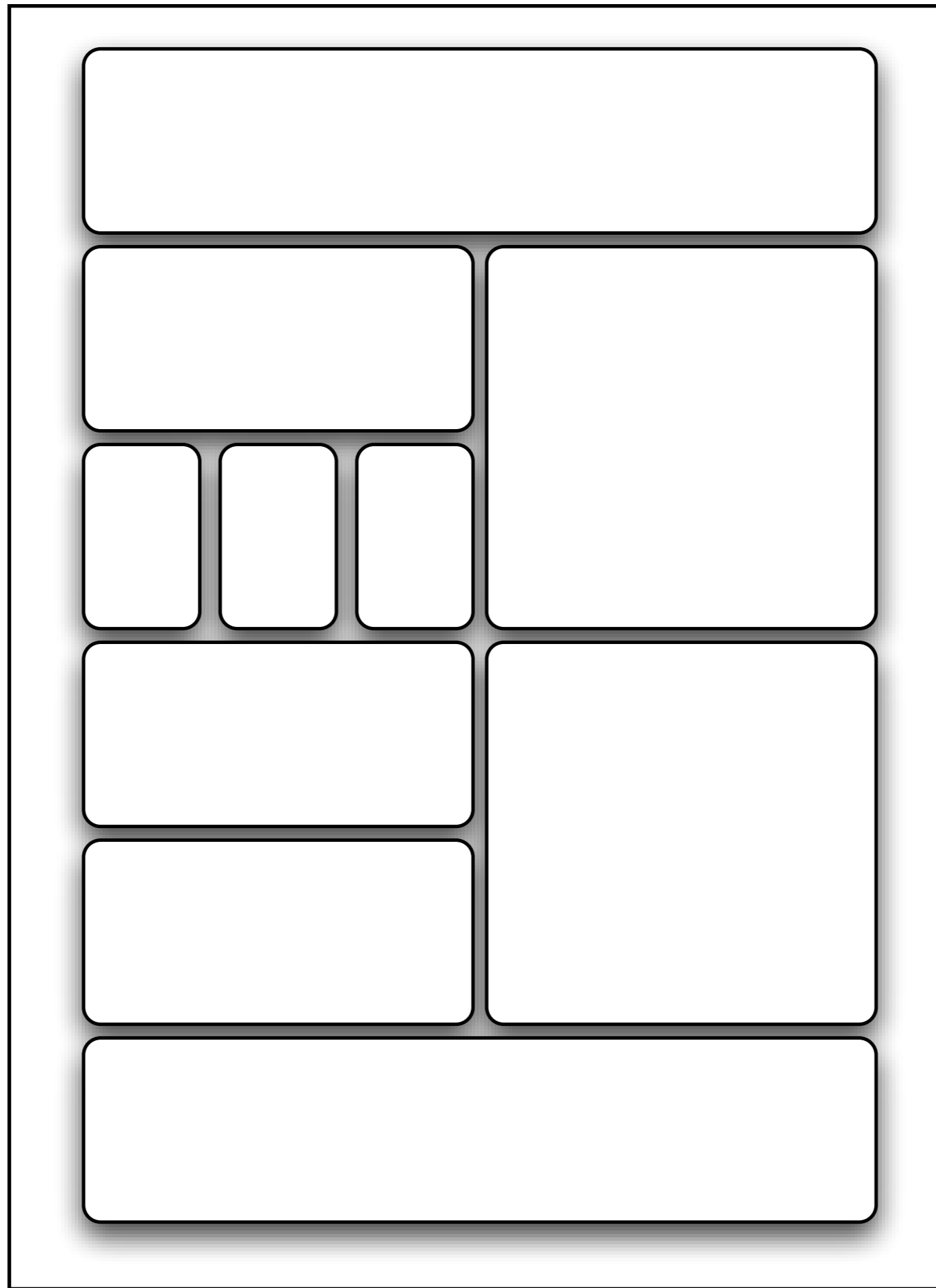
Generally, these structures form the heart of an application



Already saw the beginnings of a complex structure last Monday, here diagrams are composed of many instances of Shape subclasses

Document Example

- Document Processors Object Model
- Very simple model (but actually very powerful)
- Ignore pages etc.
- Not interested in the content (assume it is just a rectangle)



Consider a slightly more complicated example, a representation of documents

Document Model

- Document composed of:
 - Content blocks (rectangles with width and height)
 - Vertical flows (y-flows)
 - Horizontal flows (x-flows)
 - Flows can contain flows

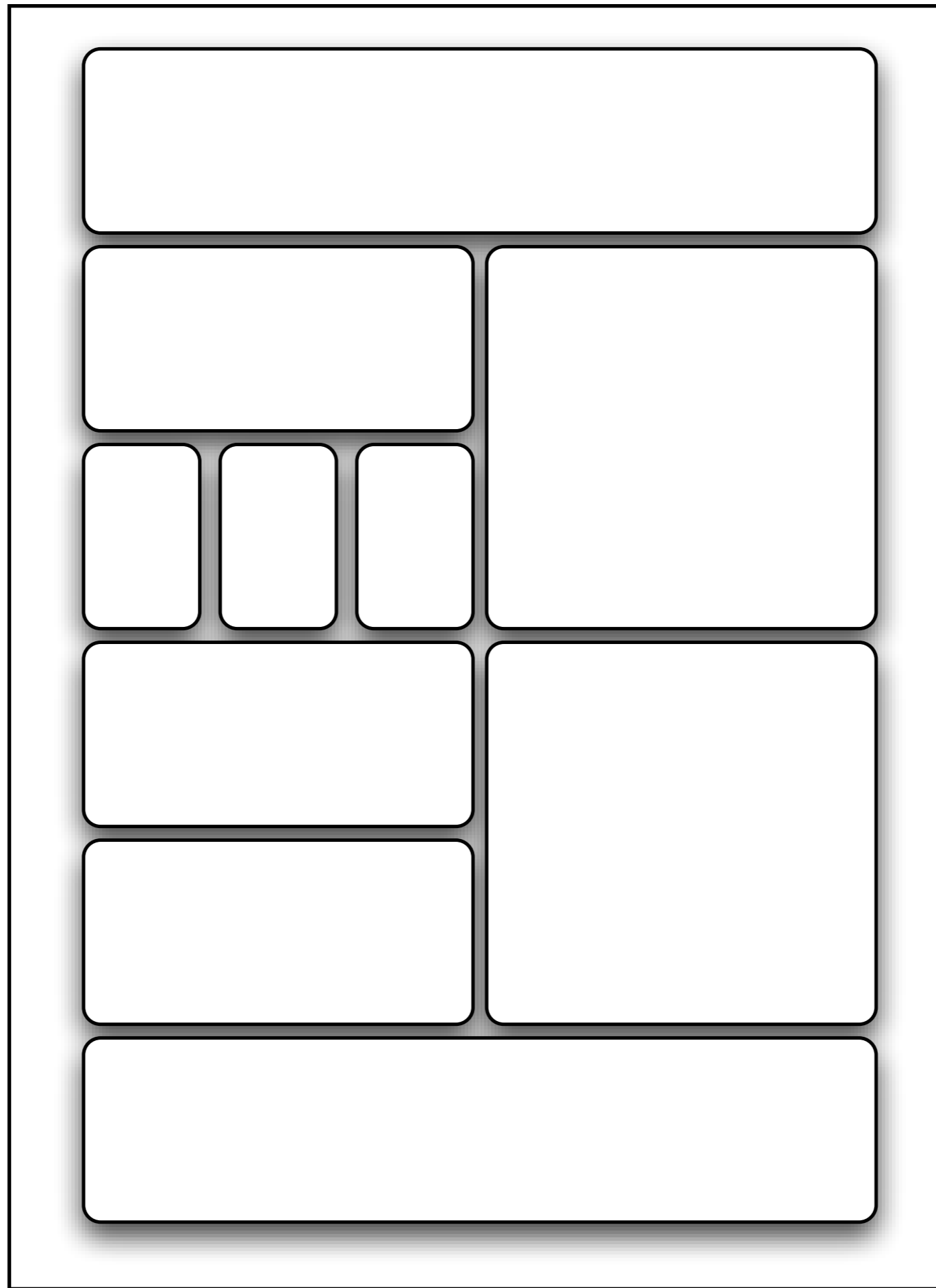
A word on flows

- A flow just lays out its children one after the other in a single direction
- Uses the width and height of its children to calculate the position of its children

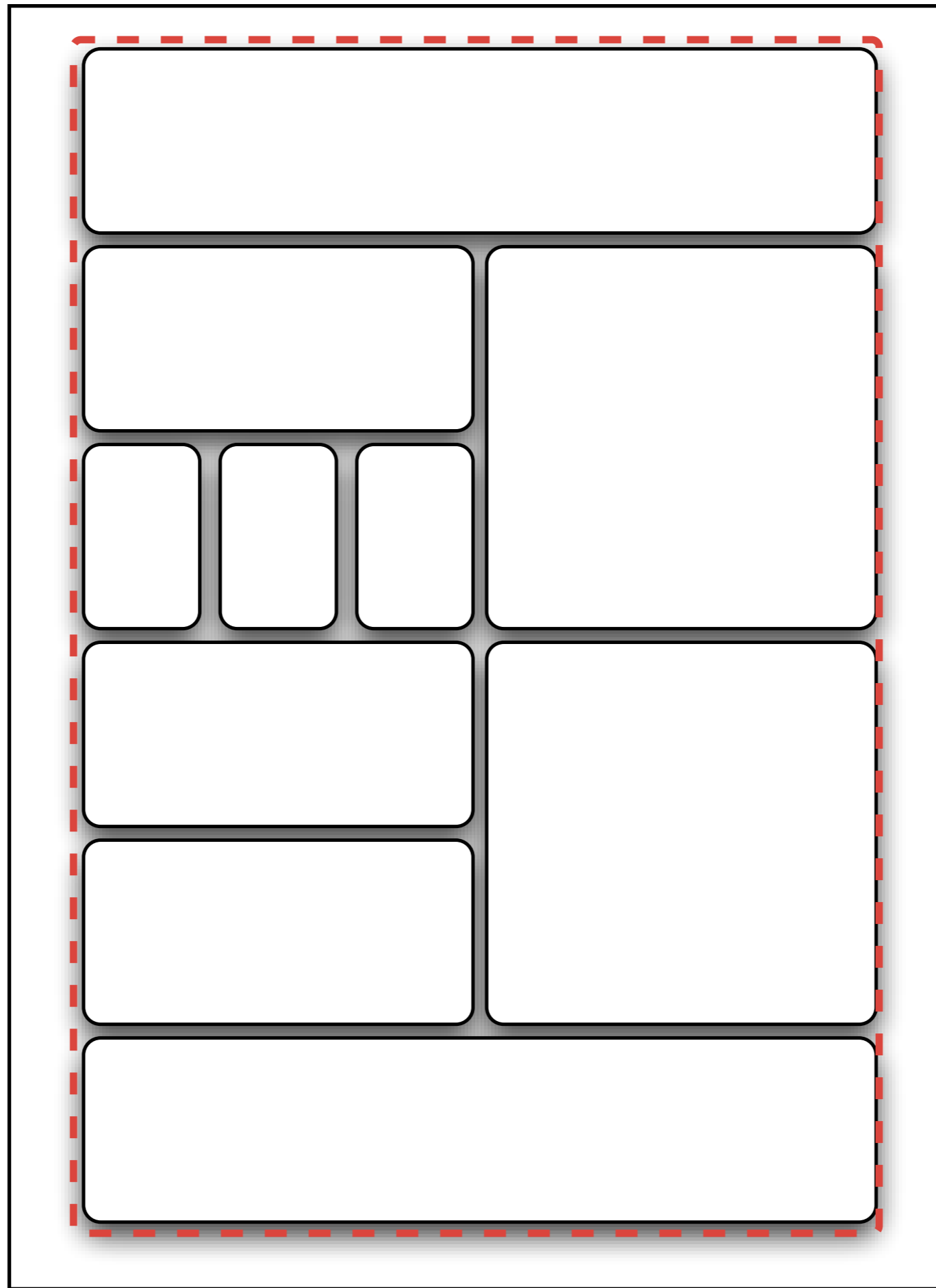
Object Representation

- Three types of object:
 - Content
 - x-flow
 - y-flow } Containers
- Need a shared base-class

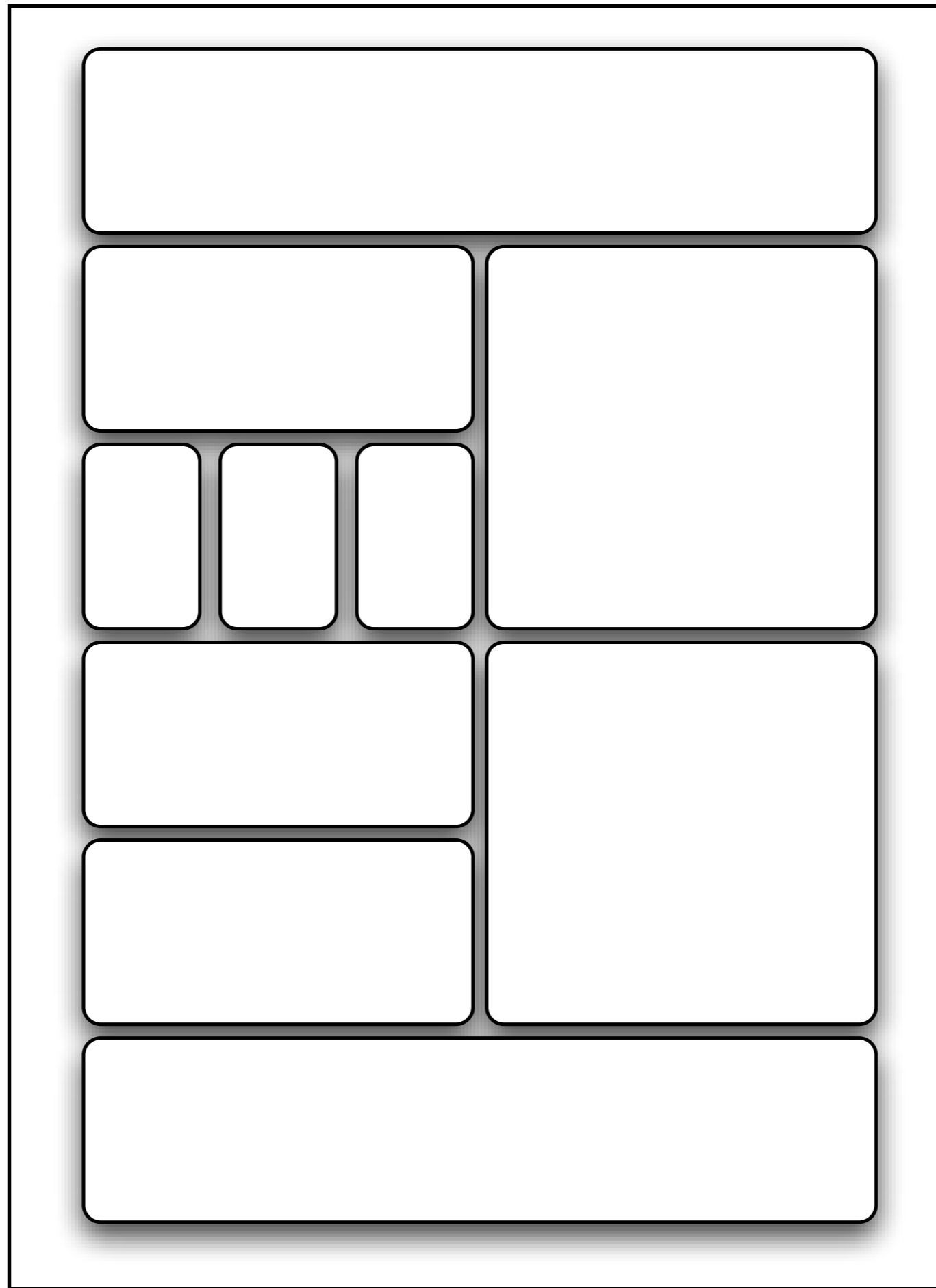
Since our x-flow and y-flow objects can contain both Content and other flows we need to derive all the objects



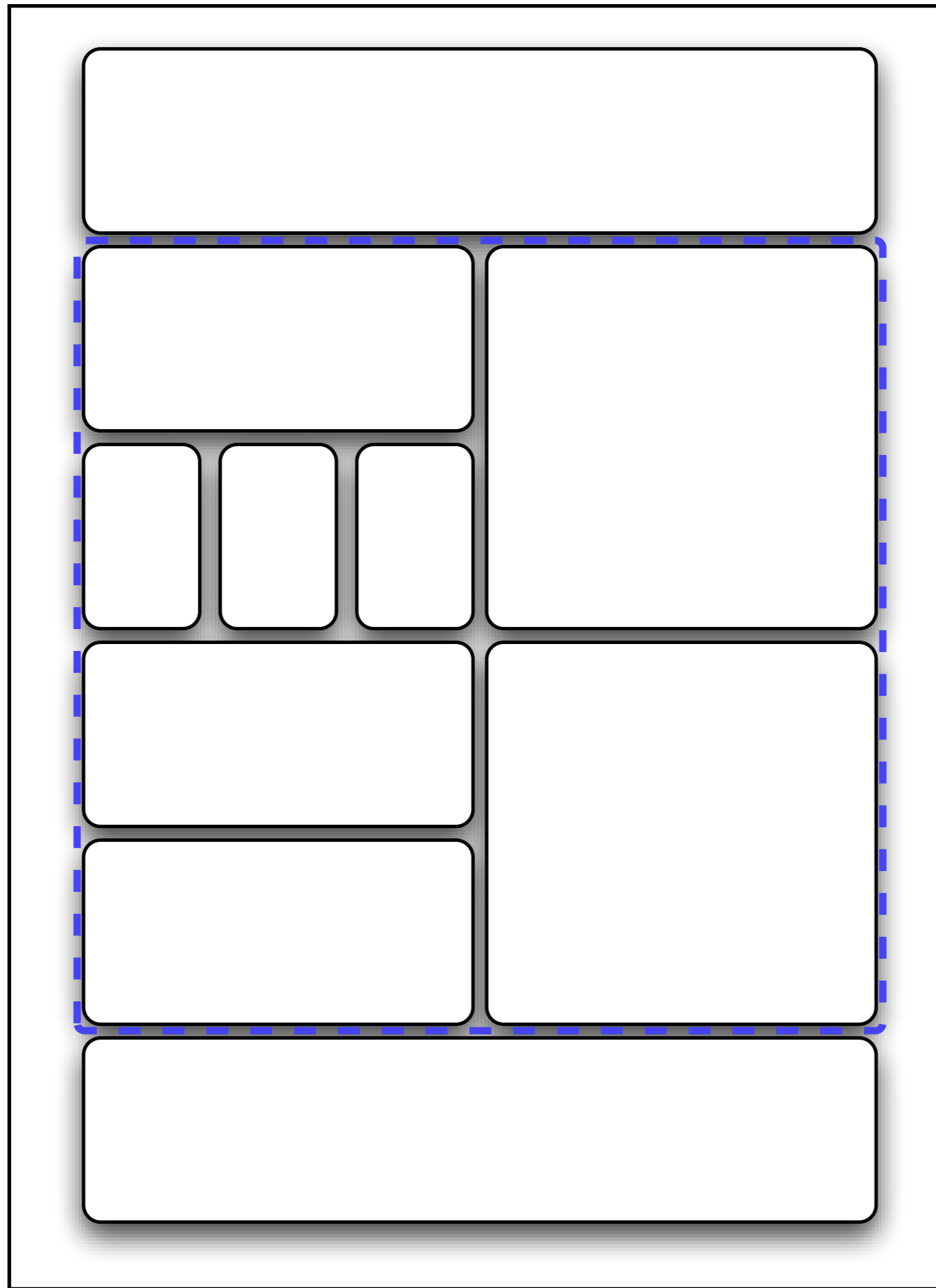
Highlight the x-flows and y-flows. Yflows in red, xflows in blue



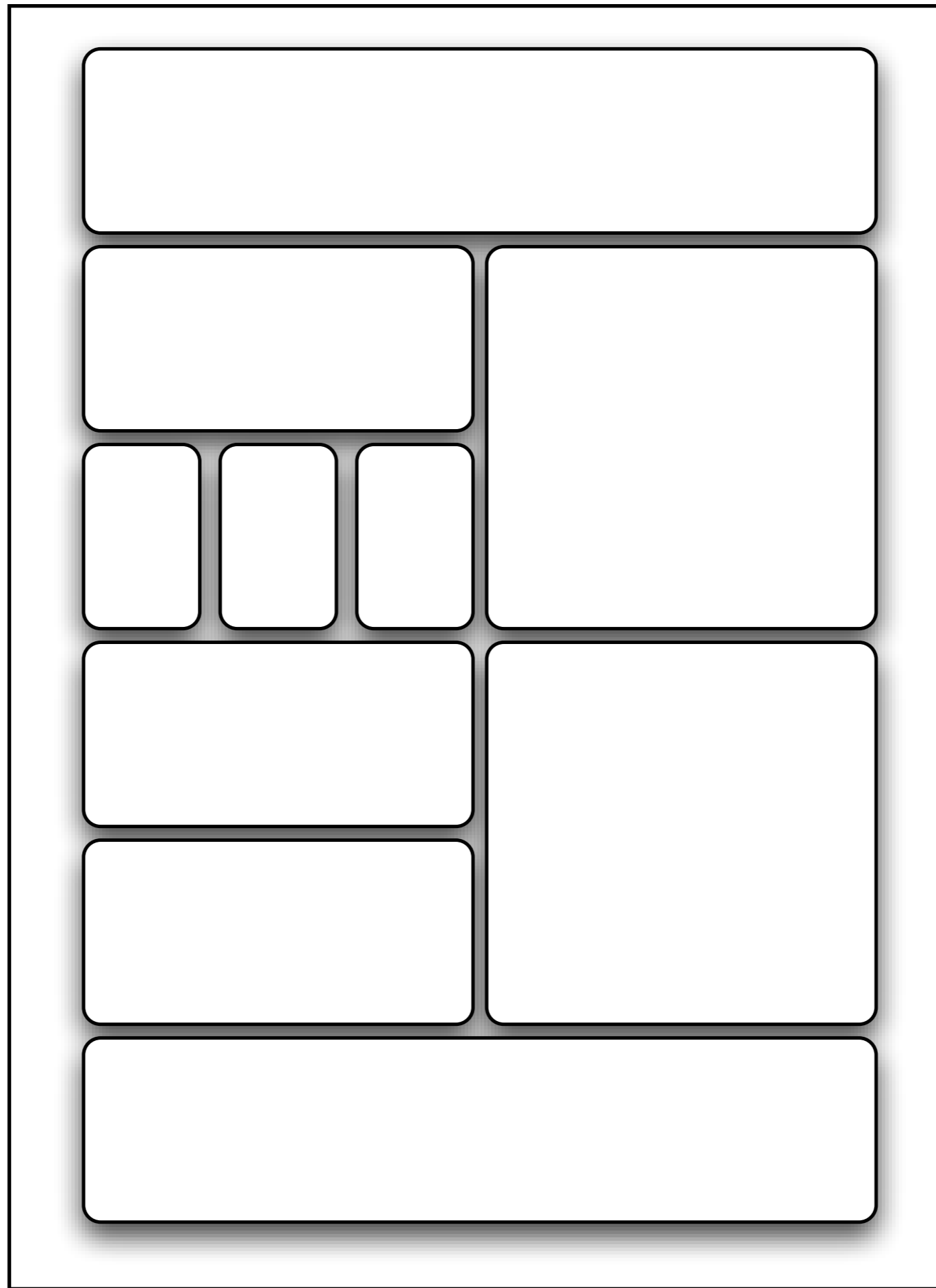
Highlight the x-flows and y-flows. Yflows in red, xflows in blue



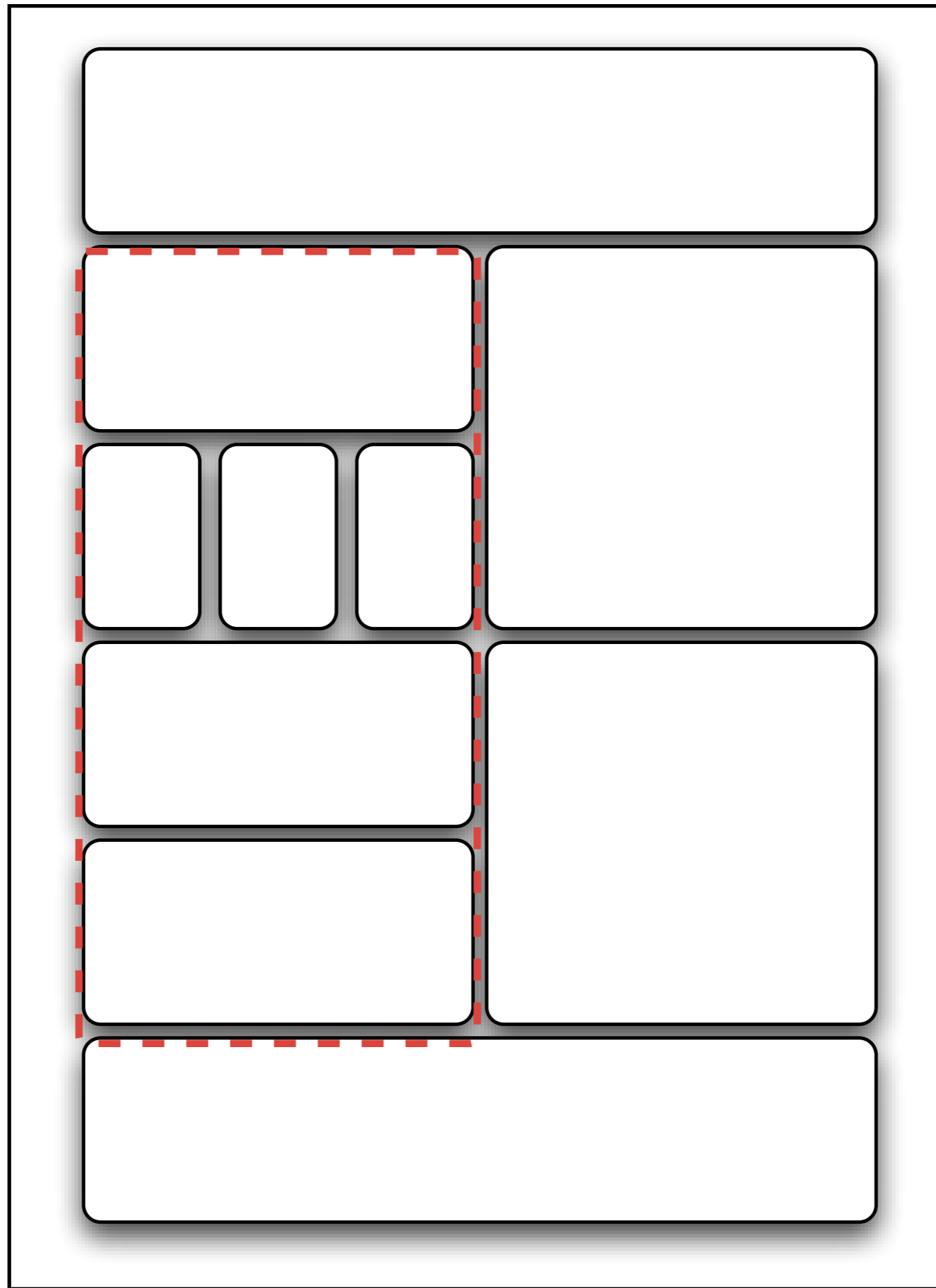
Highlight the x-flows and y-flows. Yflows in red, xflows in blue



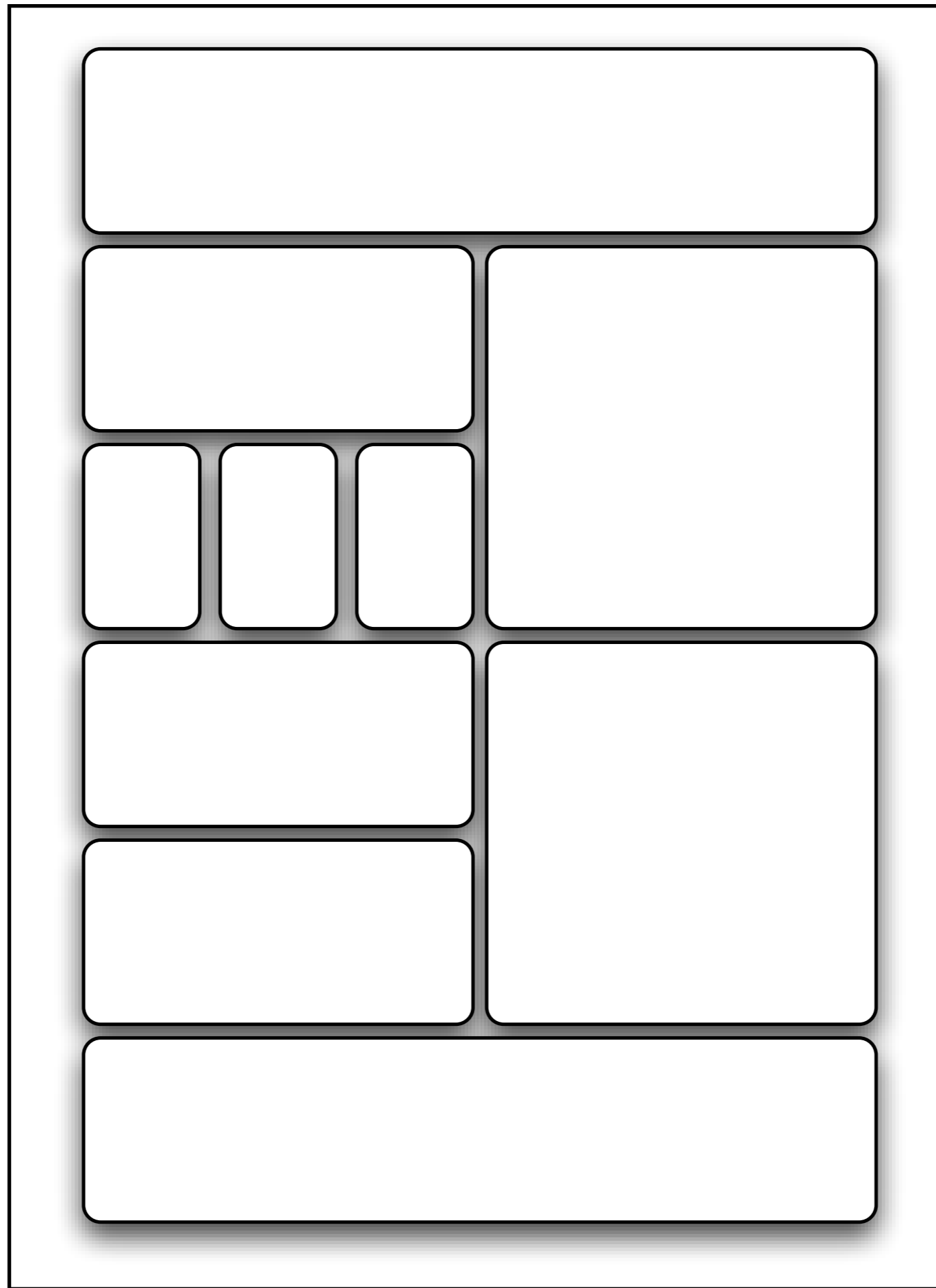
Highlight the x-flows and y-flows. Yflows in red, xflows in blue



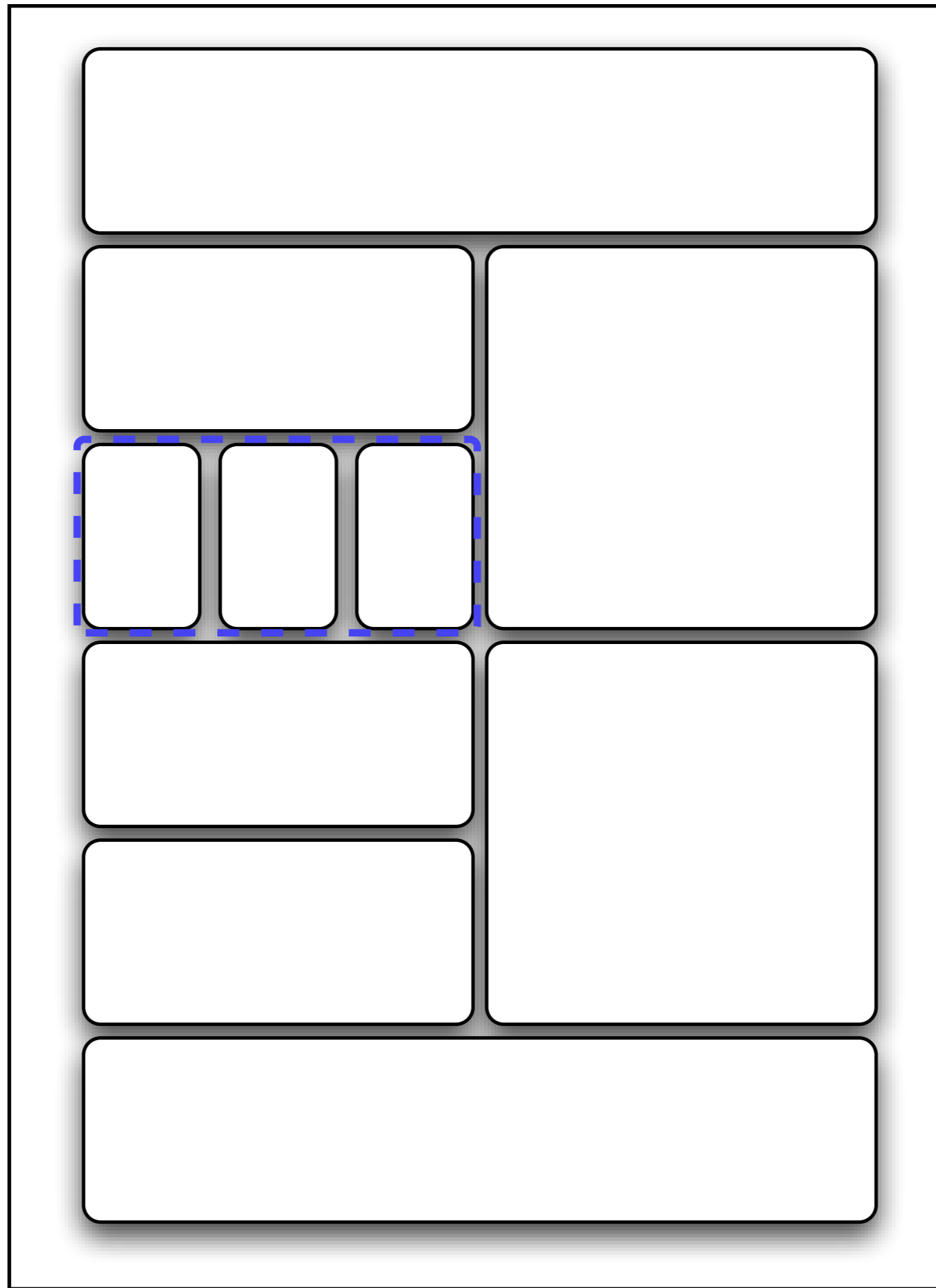
Highlight the x-flows and y-flows. Yflows in red, xflows in blue



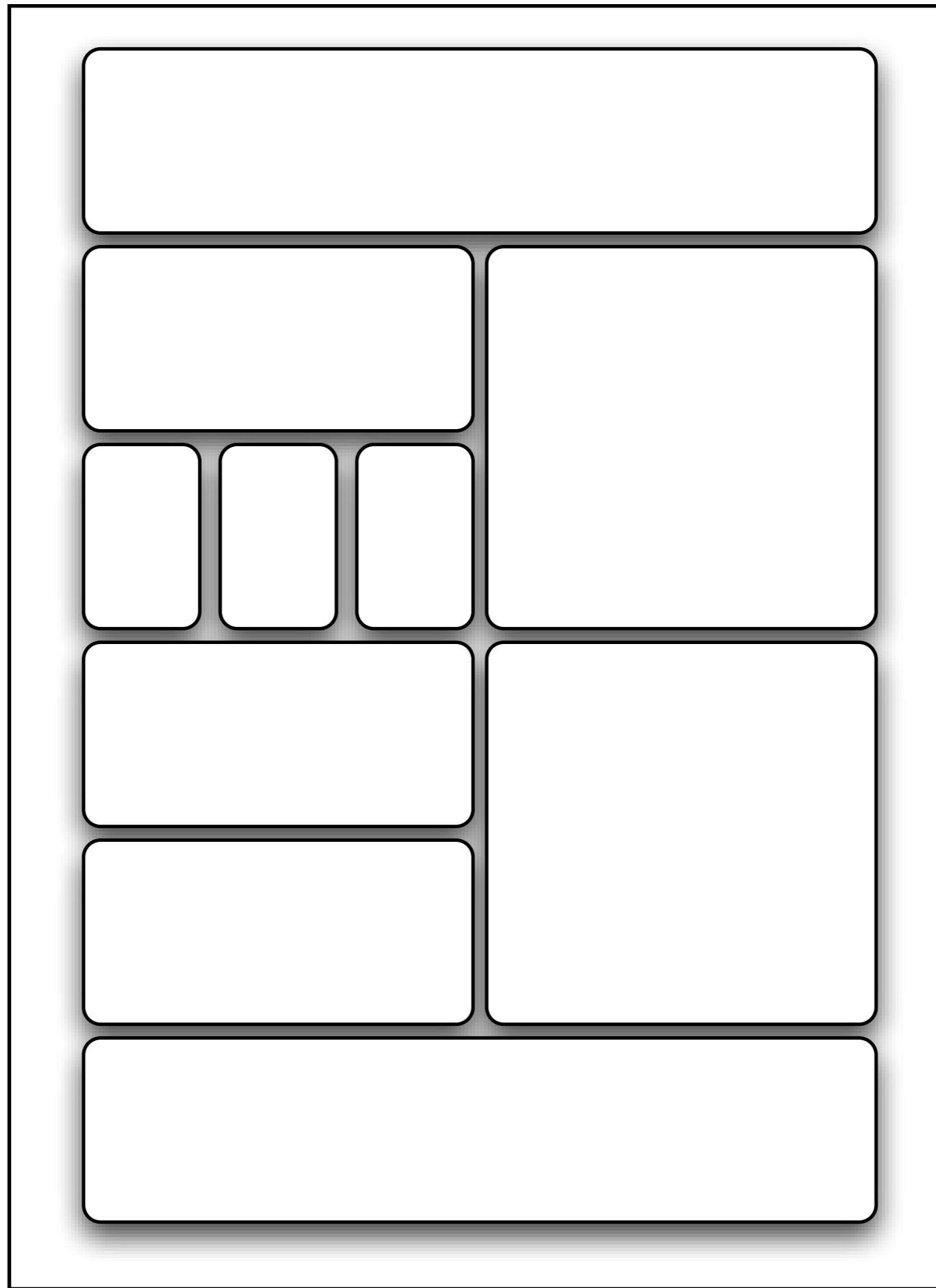
Highlight the x-flows and y-flows. Yflows in red, xflows in blue



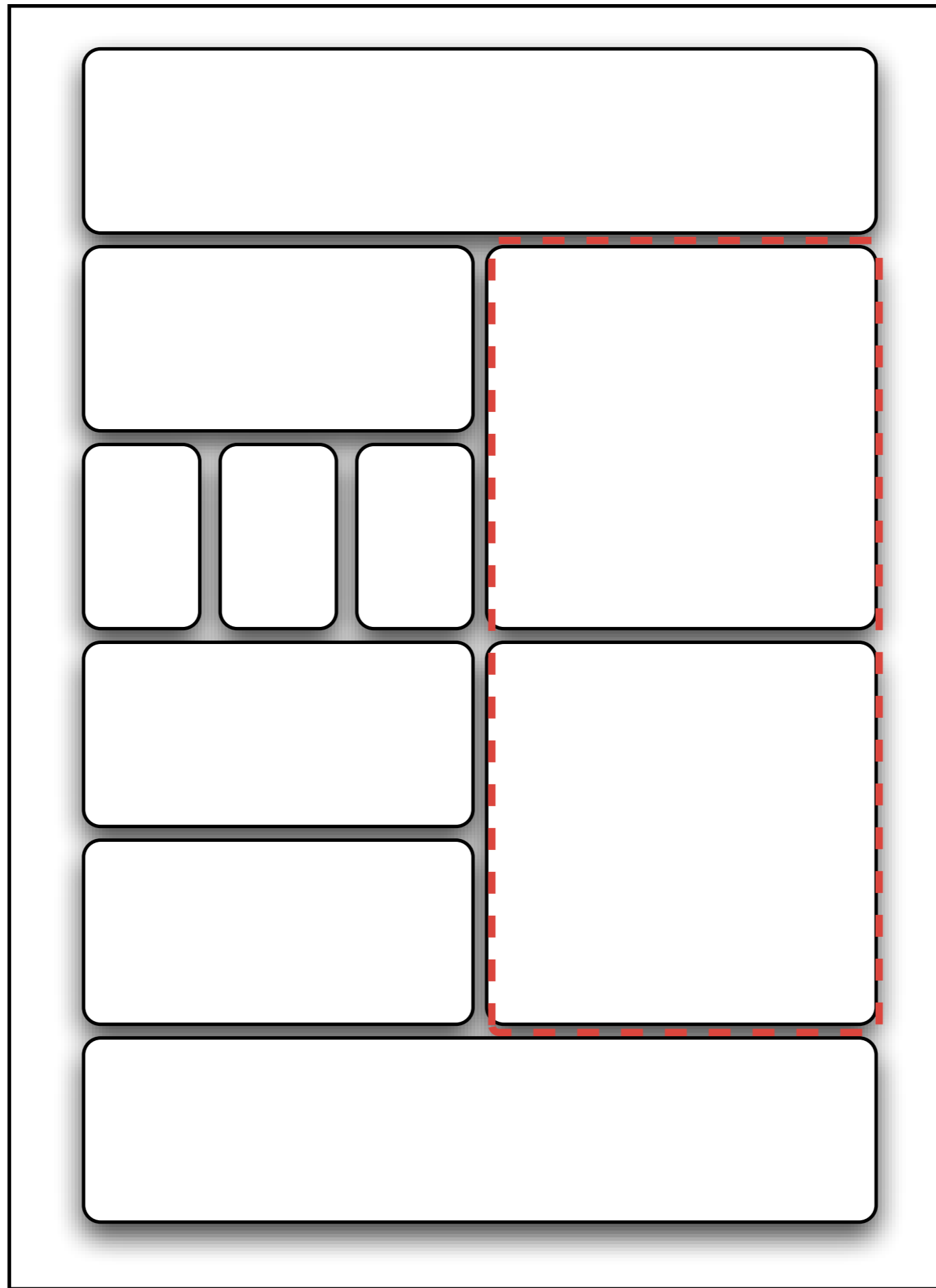
Highlight the x-flows and y-flows. Yflows in red, xflows in blue



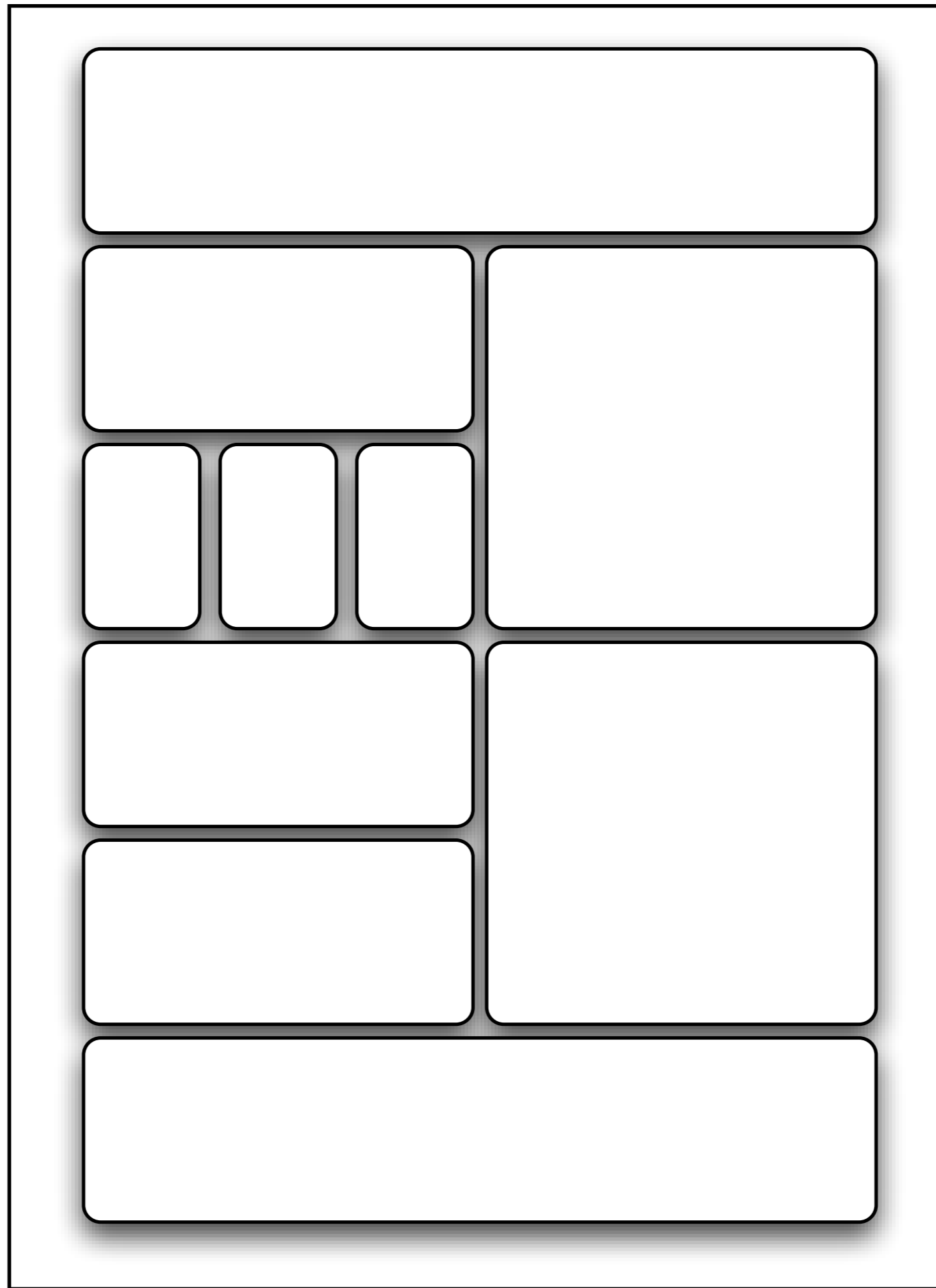
Highlight the x-flows and y-flows. Yflows in red, xflows in blue



Highlight the x-flows and y-flows. Yflows in red, xflows in blue



Highlight the x-flows and y-flows. Yflows in red, xflows in blue



Highlight the x-flows and y-flows. Yflows in red, xflows in blue

Base-class

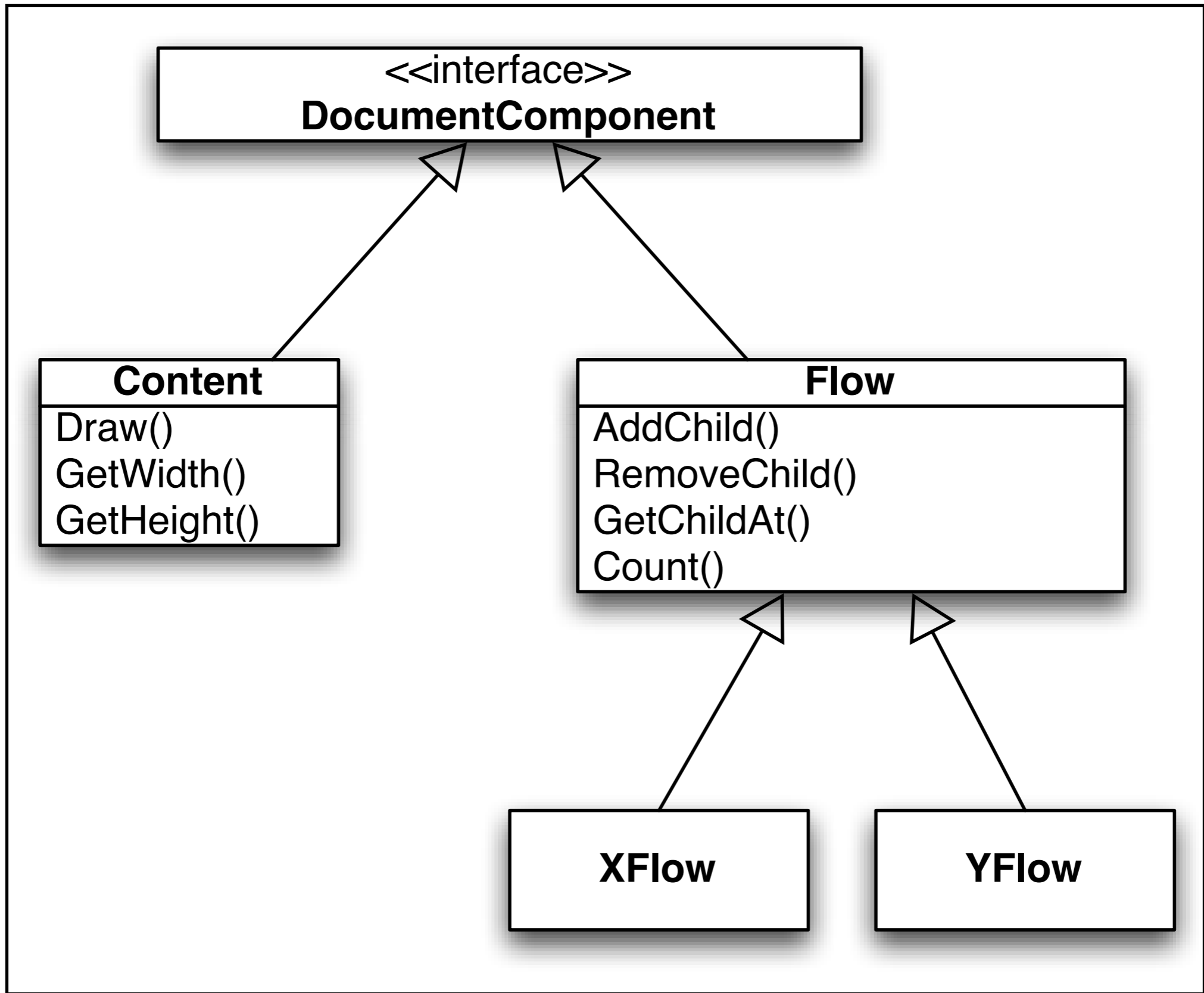
- What goes in the base-class?
- Common features between the flows and the content
- Effectively nothing
- More on this later...

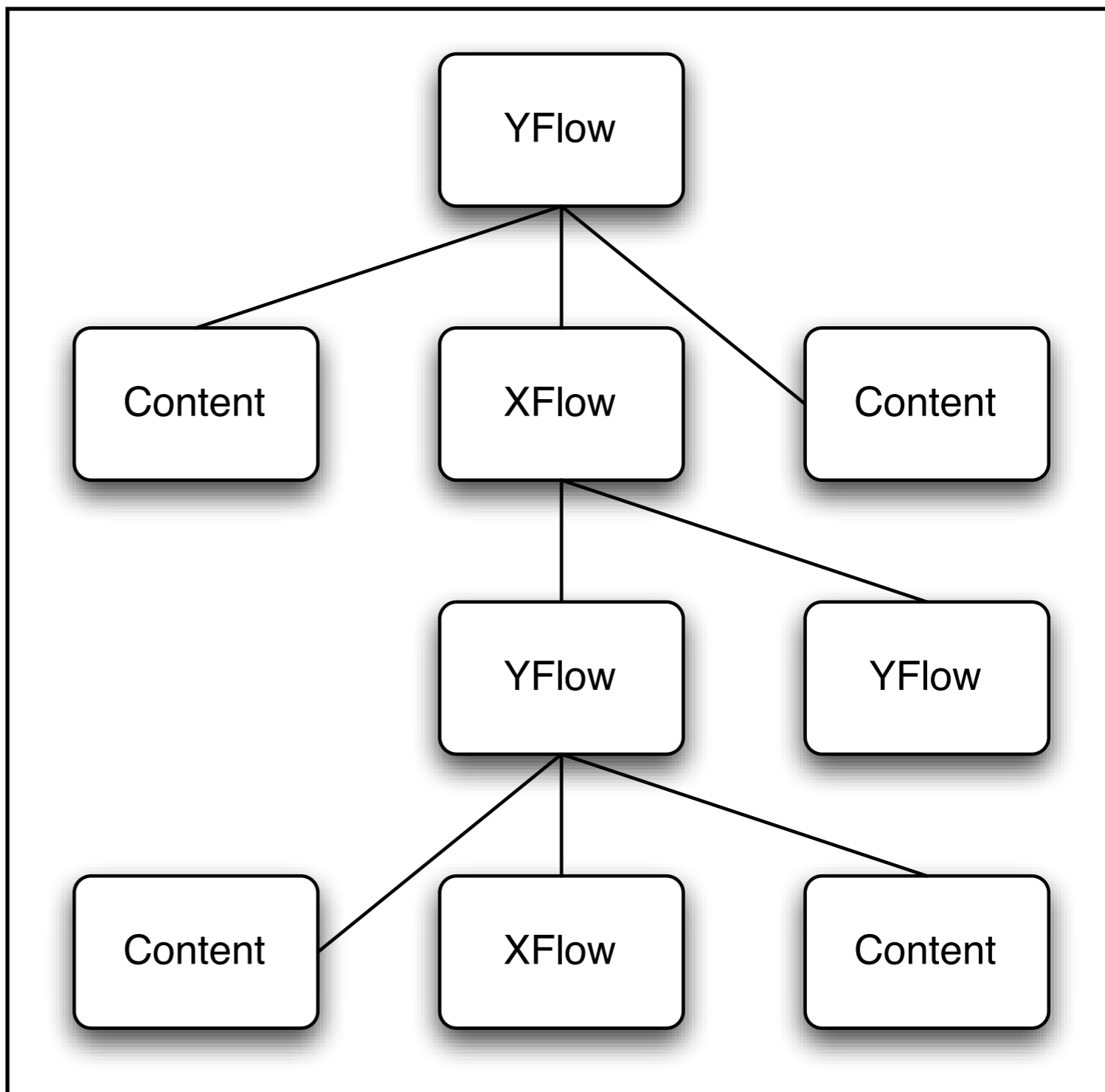
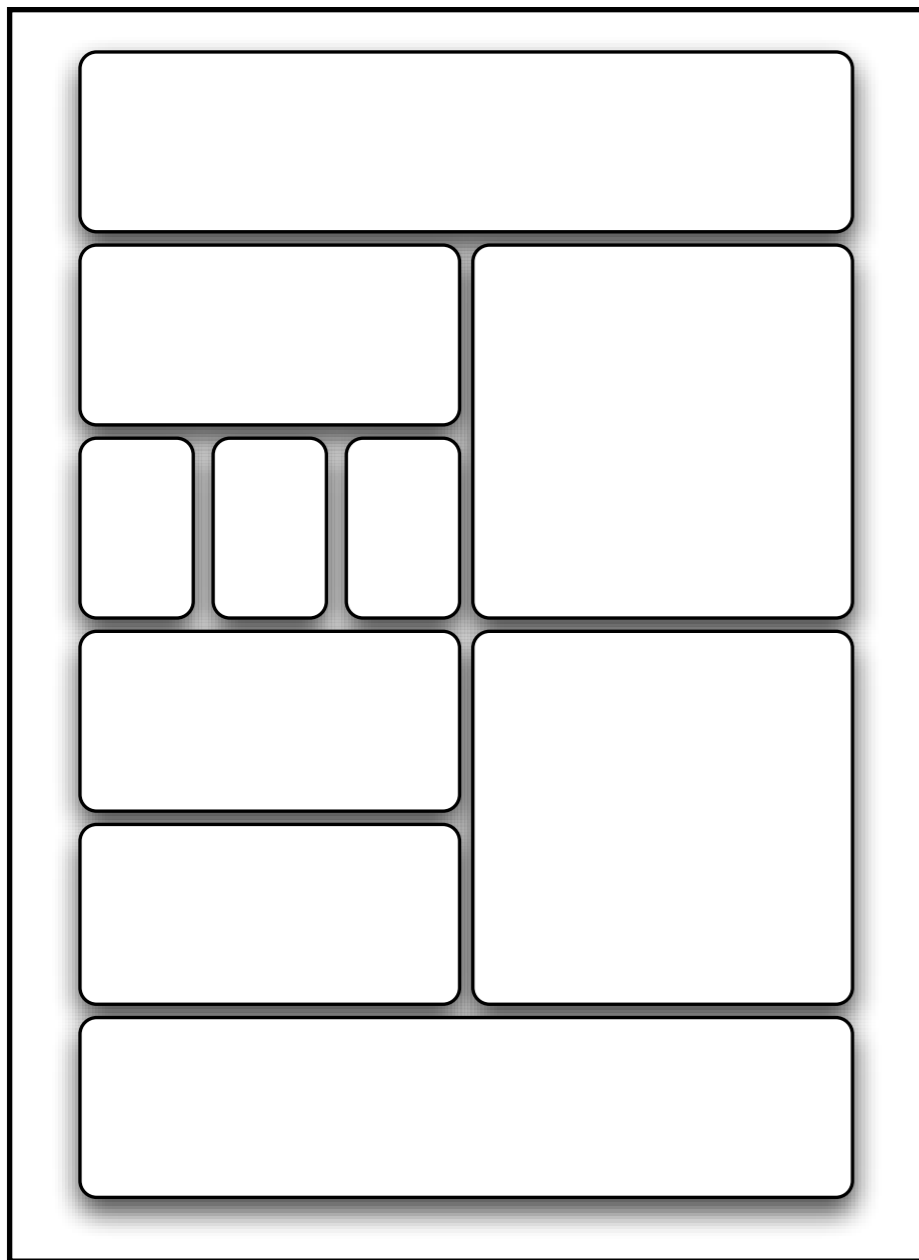
Content Class

- Represents a rectangle
- Has width and height — need accessors
- Implements Draw ()
- Not much else it can do

Flows

- Container, so need methods to:
 - AddChild
 - GetChildAtIndex
 - RemoveChild
 - Count of Child





Using a Document

- How to draw a document:
 - Get first object
 - If Content draw it,
 - otherwise process each child of the flow

Using a Document

- Suppose the first child is a *y*-flow then for each child:
 - If Content:
 - draw it at current *y*-position
 - Increase *y*-position by height of content
 - If flow, process the flow and update *y*-position

Updating *y*-position after a flow is complicated (depends on whether it is an *x* or *y*-flow etc

Similar code needed to calculate the width of a flow

Bad Design

- Proposed Design is very bad
- Any code that uses it has to understand the underlying model and objects
- We should strive to decouple client code from the objects internals
- Effectively the proposed design is no better than a procedural implementation

Improving the Design

- Be able to treat both containers and content as the same thing
- Drawing a flow should be identical to drawing a bit of content
- There's a pattern to solve this...

Composite Pattern

- ***The Composite Pattern*** allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual and collections of objects uniformly.

Composite Document

- Content and Flow share a base-class
- So they can fit in the same typed container
- This is the mistake
- Client should be able to treat Flow and Content identically
- Rework DocumentComponent interface

<<interface>>

DocumentComponent

Draw()

GetWidth()

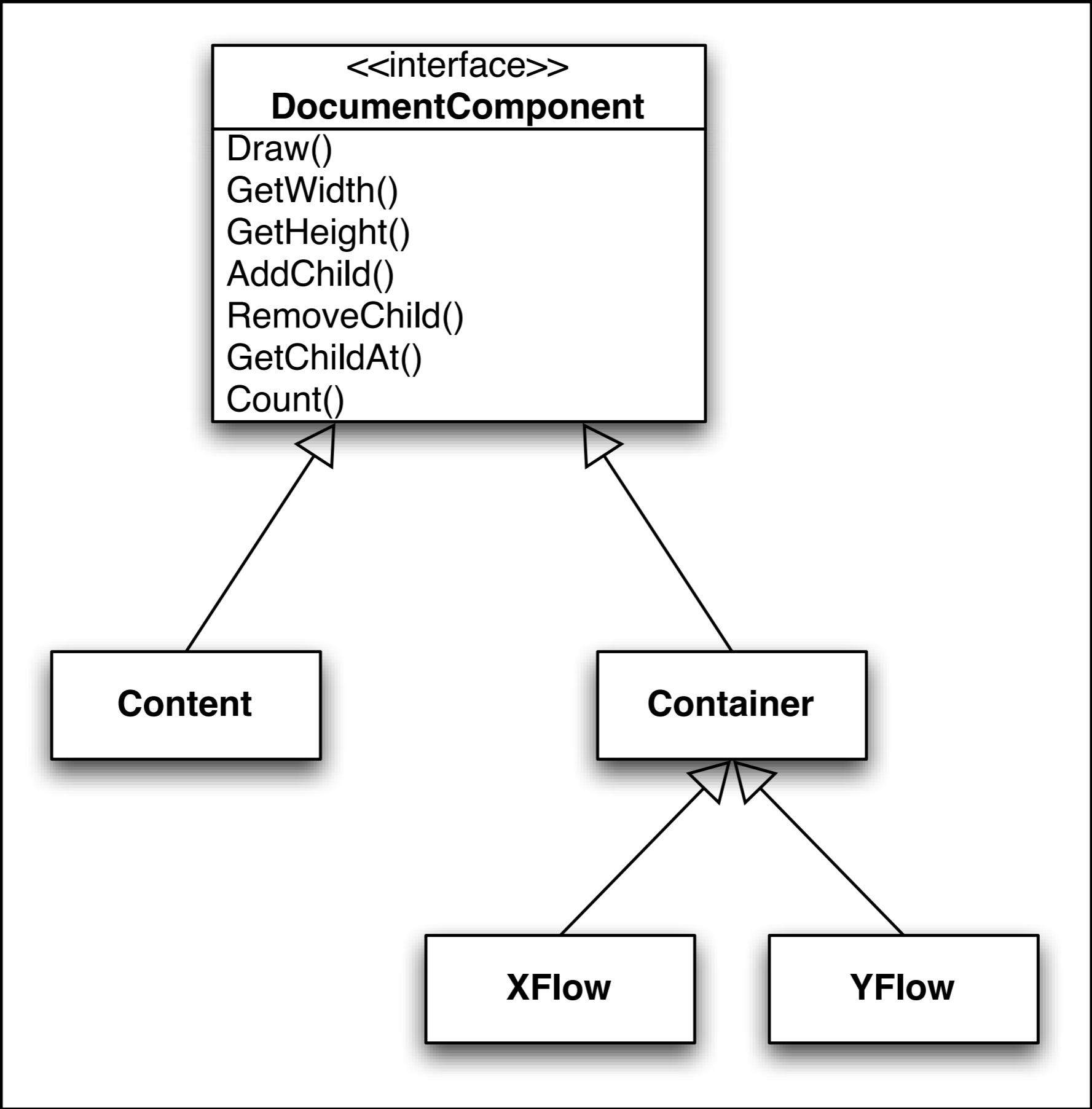
GetHeight()

AddChild()

RemoveChild()

GetChildAt()

Count()



Implementing Content

- Content implements `Draw()`, `GetWidth()` and `GetHeight()` as before
- But since it is a *leaf node* in the tree, the other methods don't make sense
- So it just provides dummy implementations
- Returns zero for `Count()` etc.

Probably also need to find a way of telling the client that `AddChild` failed... Maybe throw an exception?

Implementing Flows

- Still has to manage its children (although we've moved this into a shared class)
- `Draw()` draws its children in the correct position – previously the client's job
- Also calculates the aggregate width/height
- Note that it does not have to check the type of its children

So what have we done?

- Given both primitives and containers identical interfaces
- Client (and container!) abstracted away from the type of object
- Can still treat them individually (if it needs to e.g. adding an item to a flow)

```
YFlow *root = new YFlow();
XFlow *colFlow, *xflow;
YFlow *yflow;

root->AddChild(new Content(524, 144));
colFlow = new XFlow();
root->AddChild(colFlow);
root->AddChild(new Content(524, 144));

yflow = new YFlow();
yflow->AddChild(new Content(300, 100));
xflow = new XFlow();
xflow->AddChild(new Content(72, 72));
xflow->AddChild(new Content(72, 72));
xflow->AddChild(new Content(72, 72));
yflow->AddChild(xflow);
```

```
yflow->AddChild(new Content(300, 100));
yflow->AddChild(new Content(300, 150));
...
colFlow->AddChild(yflow);

yflow = new YFlow();
yflow->AddChild(new Content(300, 100));
...

colFlow->AddChild(yflow);
```

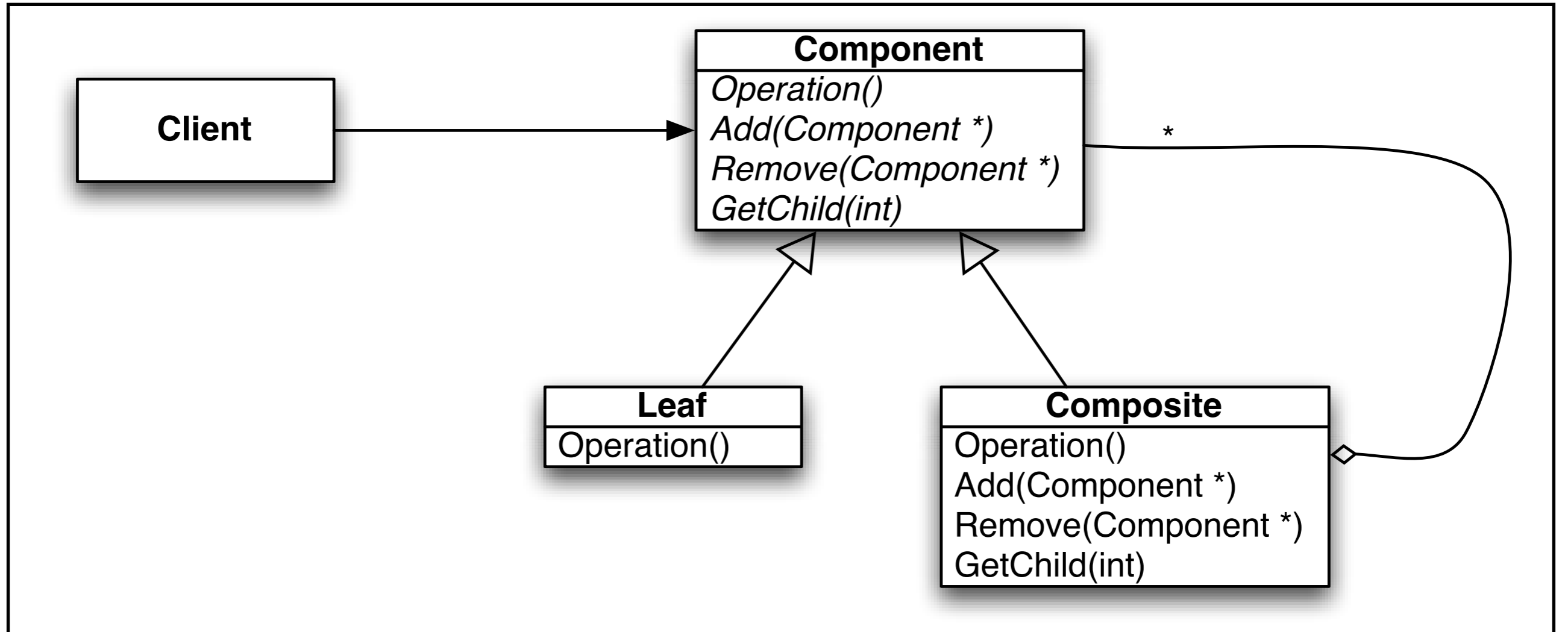
The result is that root contains our structure

Real-life Composite

- XML Document Object Model
- Elements, Attributes, Text...
- Objects to represent each of these
- Crucially, everything is also a DOMNode
- Code that navigates can do so without knowing the node's type

Composite Pattern

- Provides a consistent interface to objects in a whole-part hierarchy whether
 - They are containers
 - Or leaves



We can define a composite formally

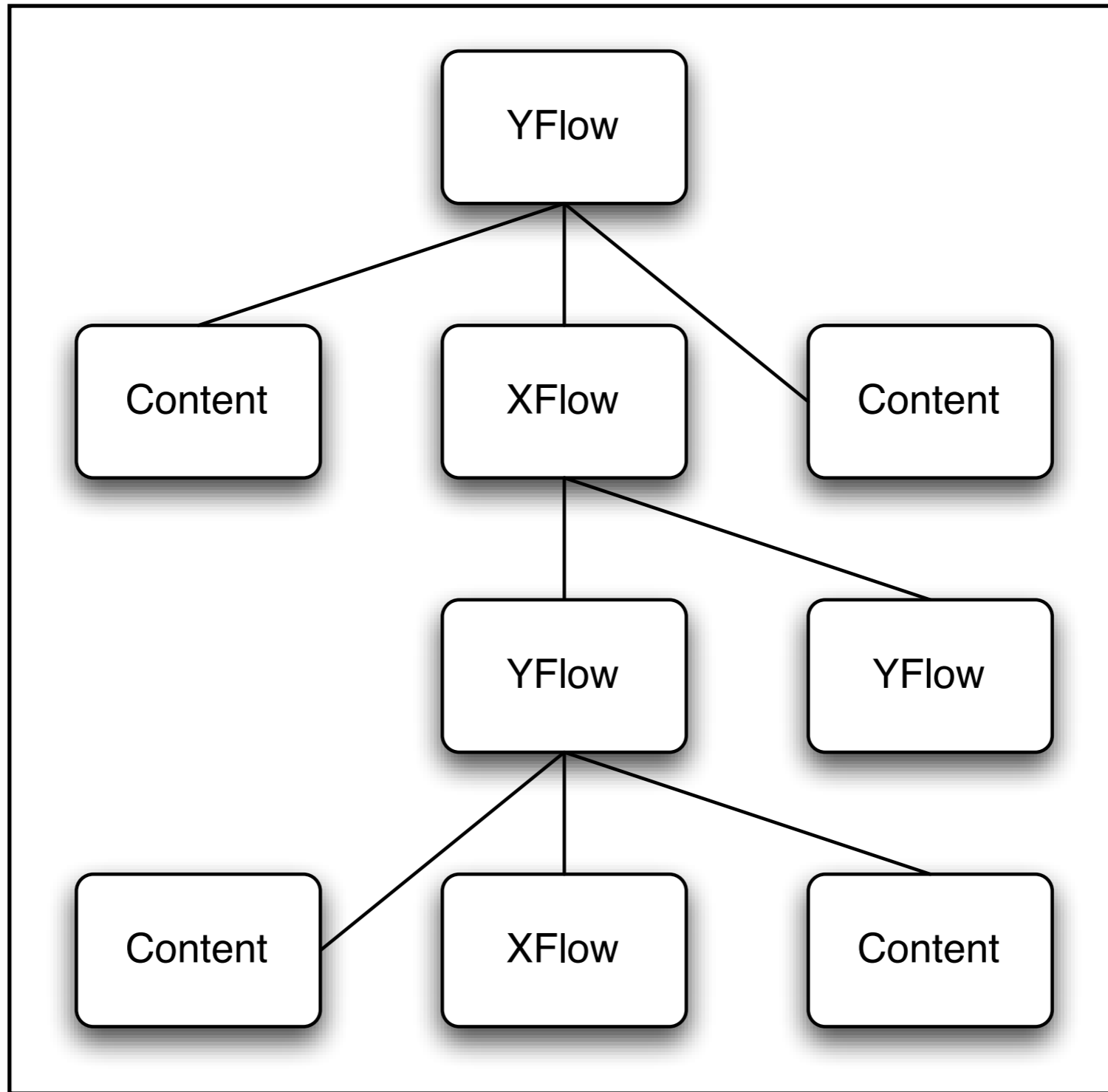
Composite Terms

- Client — Code that manipulates the object in composition uses the...
- Component — interface for all objects in the composition
- Composite — component with children
- Leaf — primitive component

Composite holds a set of children, those children may be other other composites or leaf elements. It is a recursive definition...

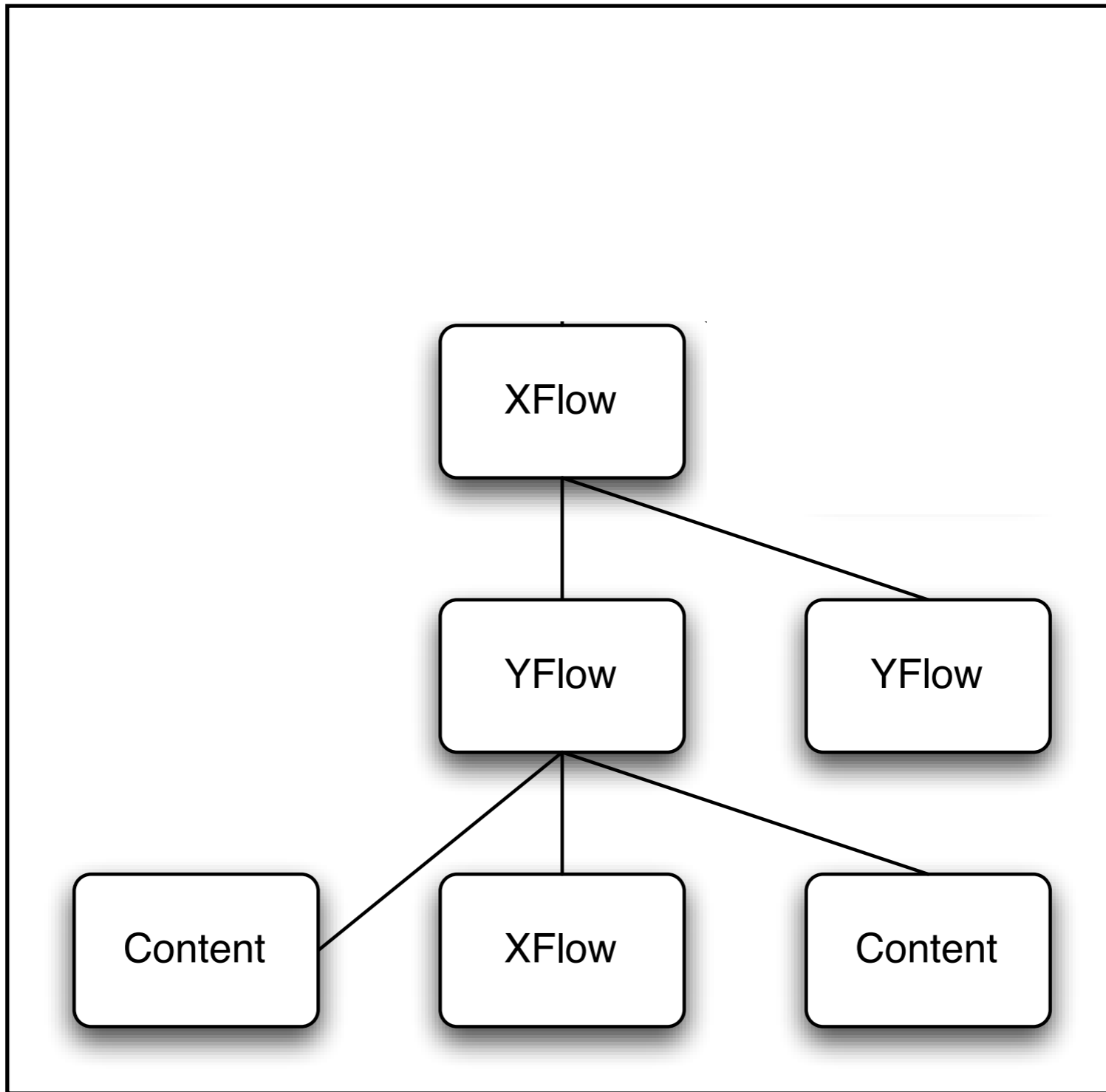
Part-Whole Hierarchy

- Means we can treat composition of objects as if they were a single part
- And also treat single parts as if they were a composition
- So we draw a flow, just as we draw some content



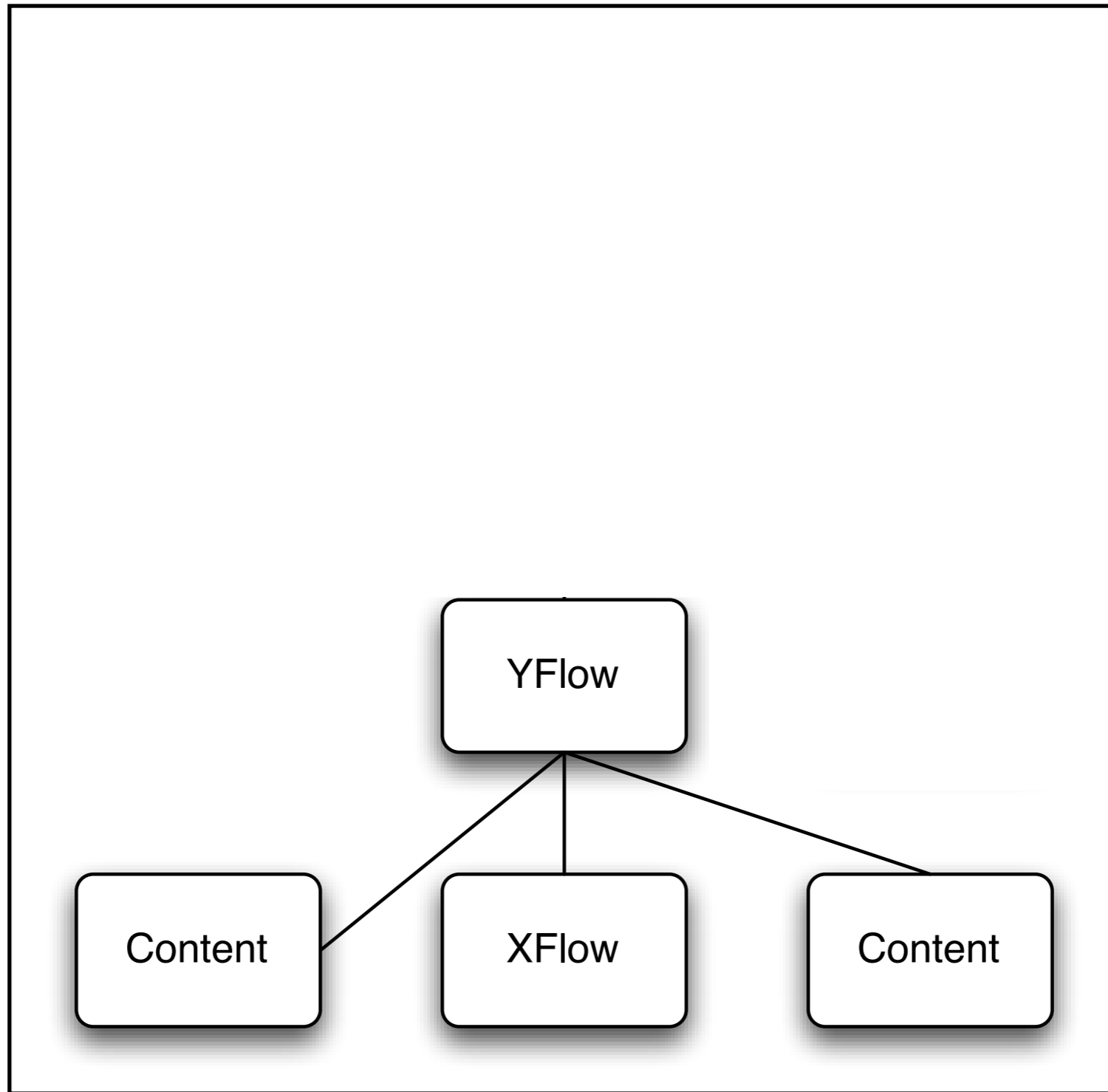
Part-whole hierarchies

Means we can treat individual parts as if they were the whole thing



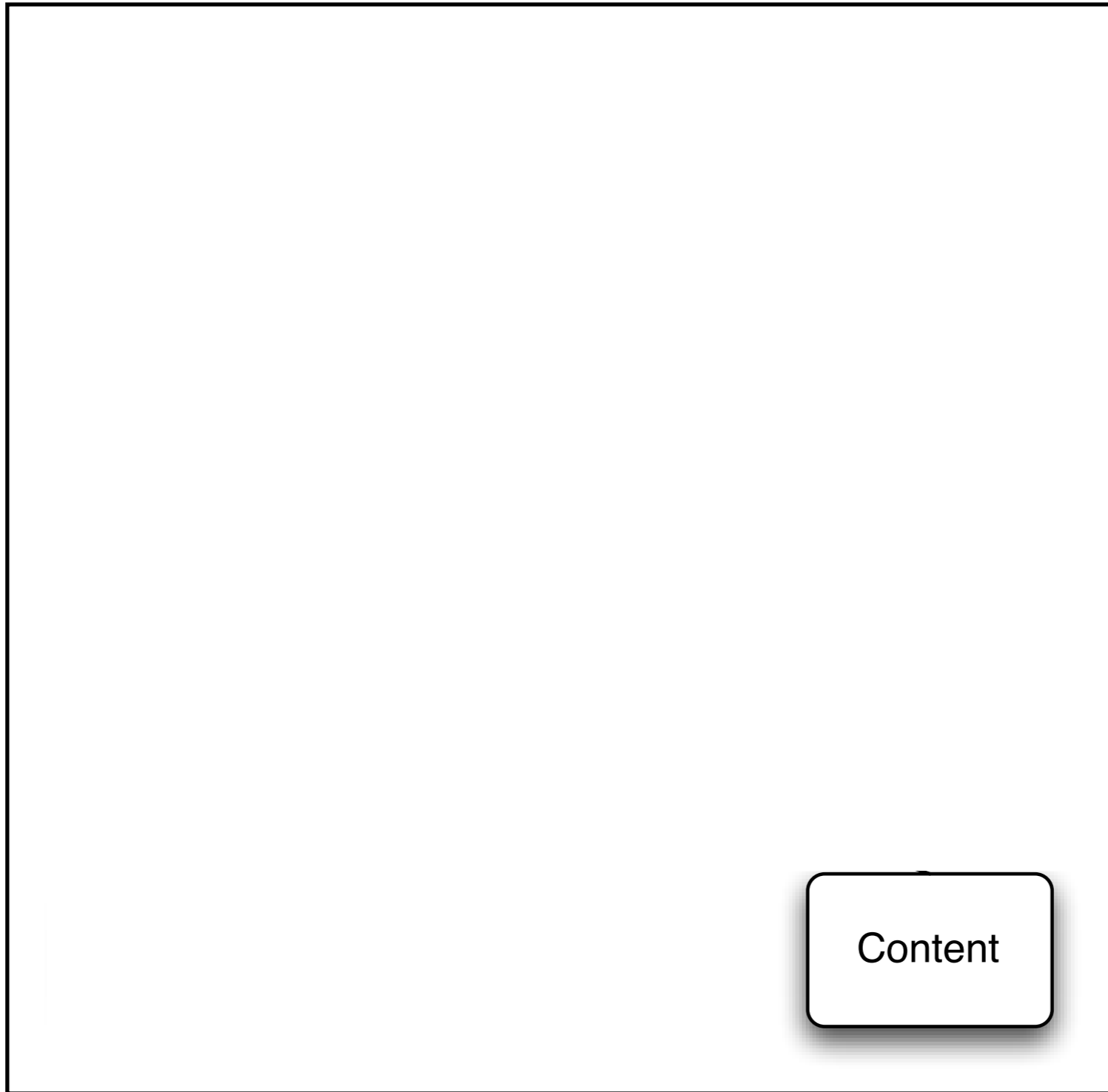
Part-whole hierarchies

Means we can treat individual parts as if they were the whole thing



Part-whole hierarchies

Means we can treat individual parts as if they were the whole thing



Part-whole hierarchies

Means we can treat individual parts as if they were the whole thing

Builder Pattern

- ***The Builder Pattern*** separates the construction of a complex object from its representation so that the same construction process can create different representations

Builder and Factory

- Builder != Factory
- Factory creates single objects
- Builder makes complex object structure

Example

- Parse an RTF document and convert into another format (text, troff, TeX etc)
- Two main tasks
 - Parse RTF structure
 - Translate this into the other document structure

Change

- Remember Change...
- If the two tasks are intermingled, then it is difficult to change the code later
- What if we want to parse Word files? What if we need to construct something else?
- As always, the solution is to abstract the two tasks apart

RTF Parser

- Parses the RTF file
- Comes across tokens informing it about various changes in the document
- Needs to cause appropriate structures to represent these to be built
- Calls methods on an interface
- The 'Director' — Directs construction

Document Builder

- Responsible for building the object structure
- Told about various changes that have occurred
- Builds appropriate structures as instructed
- Implements methods on the interface
- The Builder — builds the actual objects

Builder Complex

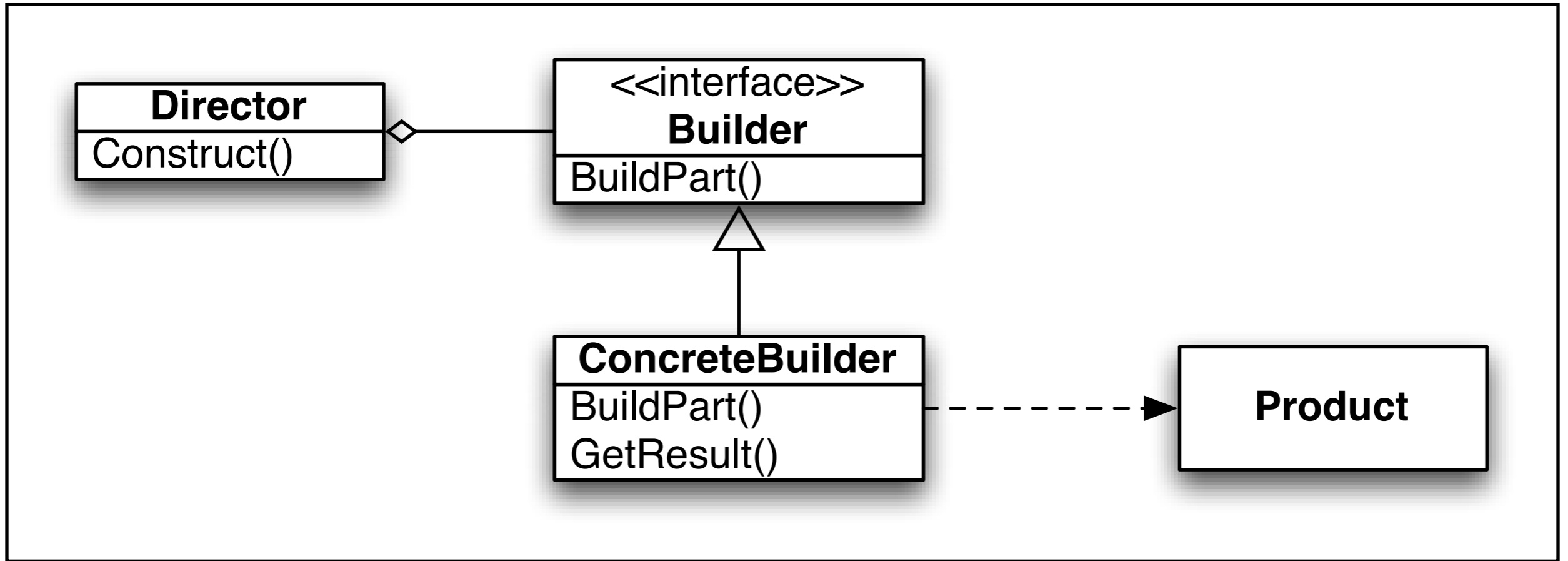
- Builders abstract complex construction
- Imagine building a table
- Probably have methods
 - `BuildTable()`
 - `BuildRow()`
 - `BuildCell()`

Table has rows, rows have cells, cells can contain tables etc

Build Table causes a new table to be built, Build Row causes a new row to be built (needs to know which table it is being built in), Build Cell causes a new cell to be built (need to know which row it is being built-in)

Decoupling

- Director (Parser) and Builder are now totally decoupled
- Separated by the interface
- Can change the Director for a new one, providing it uses the same interface
- Can change the Builder for a new one, providing it implements the same interface



Builder Terms

- **Product** — represent the complex object being constructed (more than one object!)
- **Builder** — specifies an abstract interface for creating parts of a Product object
- **ConcreteBuilder** — Responsible for building a specific product
 - Provides a method to get final product

Builder Terms

- **Director** — Uses a Builder interface to direct the construction of the complex object

```
Document *document;  
Builder *builder = new  
DocumentBuilder();  
  
RTFParser *director = new RTFParser;  
  
parser->SetBuilder(builder);  
  
parser->Parse(someFile);  
  
document = builder->GetDocument();
```

Using Builder Pattern

- Client:
 - Creates a Builder
 - Creates a Director
 - Tell Director to build object using Builder
 - Gets Object from Builder
- Client and Director can be same object

Director

- When told to initiate construction:
 - The director calls methods on the shared interface to construct various parts of the complex object
 - **NEVER** actually touches the objects it creates

Builder

- When the director calls:
 - Build the requested parts and add to the object
- Return completed object to client