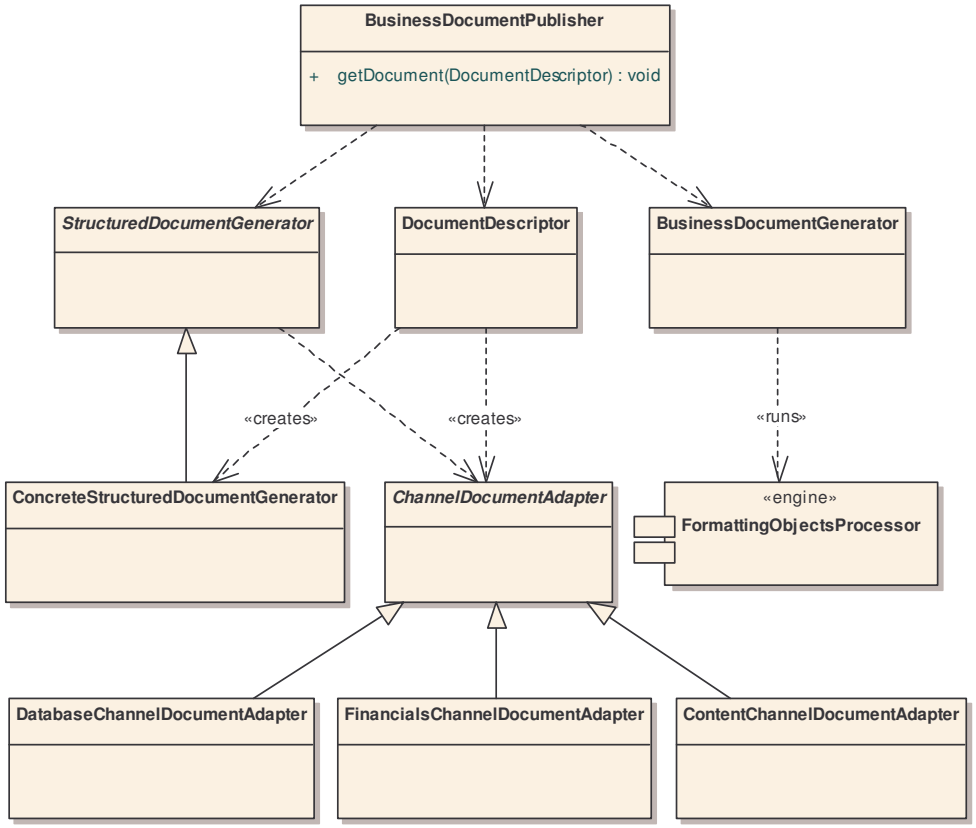
# Business Document Publisher

*Aggregates multi-channel data and content templates as input to the generation of a document that is suitable to serve as an offical statement of record.*




## *Background*

Organizations have production, reporting, financial, and other data, as well as structured content that are used as input to create business documents. The business requires an automated process to deal with the volume. Quality must be high since the documents officially represent the company and its products and services. The page layout must be precise since some sections of the document have specific visual constraints and boundaries. Open standards and open viewing tools are required for truly global consumption.

## *Value and Benefits*

The volume of documents produced by the business is too great to perform the process manually. Driving an industry-leading word processor as an automated document engine is a poor option. Such an engine is slow, imprecise, and fragile. It lacks scalability, and must be hosted on the operating system that the word processor was designed for.

A publishing process is needed that can consistently gather data, manipulate it, and put it into a highly readable and printable format. The process needs to be able to generate a high volume of documents with repeatable precision. It must be fault tolerant,

scale as business needs grow, and run on whatever enterprise platform and operating system that best supports the organization's environment. The published documents must be consumable using prolific, open tools.

When an organization is required to produce hundreds, thousands, tens of thousands, or more documents in a day, and can meet that level of service with the desired precision and consistency, there will be no complaints about document production. When the consumers of such documents find them highly readable and printable, they will be pleased with at least this part of their customer experience.

Examples of business documents include the following, many of which I know first hand:

- Billing and Premium Statements
- Contracts
- Forms
- Instruction and Procedure Manuals
- Insurance Policies and Claims
- Invoices
- Product Catalogs
- Quotation and Pricing Statements
- Student Guides

The list of possible applications literally goes on and on. Note also that there is no need for the overall pattern to be limited to generating highly chiseled documents. The four-step process highlighted next, in combination with *Styled Page (page #)* may be effectively used for Web publishing.

## *Putting It to Work*

There are four primary steps that are performed in any automated publishing process:

1. Aggregate all the data that will server as input to the document
2. Assemble the gathered data into a structured data stream that can serve as input to the page layout engine
3. Pour the input data into a page layout and content formatting engine that produces the desired print-ready document
4. Publish the document by distributing the finished product as necessary

The *Business Document Publisher* executes those steps. Each is a separate behavioral concern, and, therefore, can be examined individually as the four key strategies of this pattern.

### *Data Aggregation*

Data aggregation is the process of gathering information from multiple, disparate sources into a single object model that can be traversed in a convenient manner. Data gathering requires some sophistication because of the potential for reading data from multiple, disparate channels. In the age of enterprise information systems it is not difficult to imagine an environment in which your data channels consist of a production database, a

legacy database, an ERP application, a partner catalog, and a content management system.

It is also not difficult to imagine a much simpler "silo" application. If your domain will consistently read from a single data channel, such as a production or reporting database, then there is no reason to tackle the complexities of a multi-channel publisher. I will first present the single-channel data reader. With that behind us the multi-channel aggregator will be much easier to understand.

It is fairly common to publish documents out of a single-channel input mechanism. If you are generating a policy document or another kind of contract, the data that will make up the document content is probably in the production database of the system that is publishing the document. It may be a new system that was architected, designed, and developed to solve a single, well-defined problem. This might correspond to the Basic B2C or cookie-cutter E-Commerce B2C systems that were described in Chapter 2. The documents being produced get their data from the application's native data source.

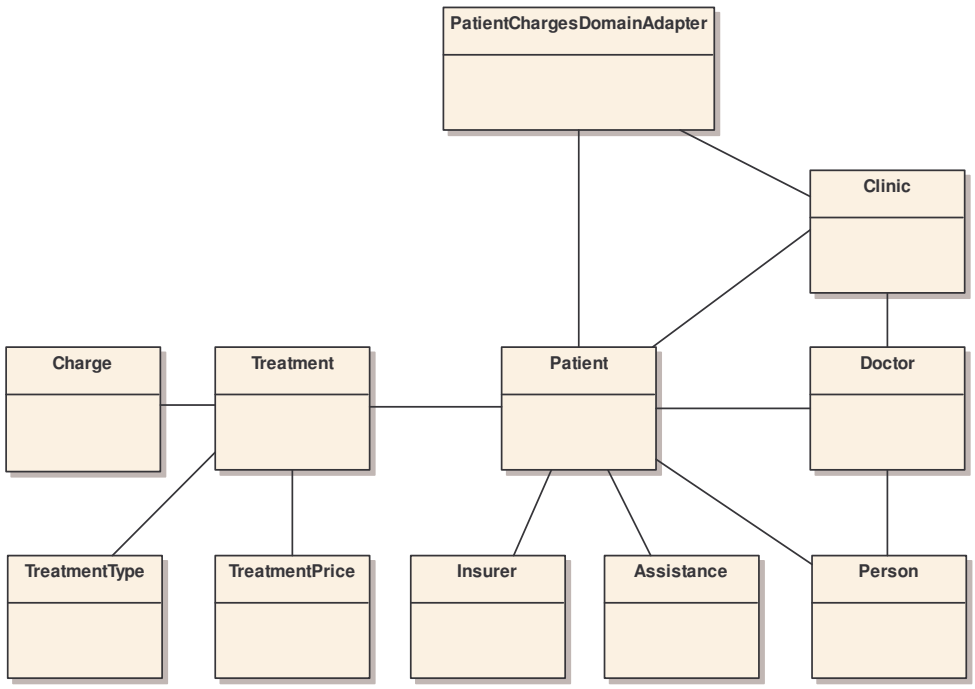
There are several patterns used to access the domain objects from a standard data source. One or more of the following[1] may be what your enterprise application already uses. There is no reason for us to search for another access pattern since our document publisher should use the one that is part of the established system architectural blueprints. So we will just use what we use elsewhere in the same system:

- Business Object
- Composite Entity
- Data Access Objects
- Domain Model
- Domain Store
- Service Data Objects
- Table Data Gateway
- Table Module

There is generally a root domain object that provides the primary, dominant information context for the published document. The dominant root context will many times have a tree or graph of domain objects that are navigable from it. There may be other extraneous root domain objects that are not naturally part of the primary root domain object. It is always convenient to access all data as if it were part of the same domain graph. Or there may be domain objects that are reachable from the primary root context but that we would rather access without having to navigate to it ourselves using the available route. In such cases it would be nice to take a shortcut. Or we may not have a natural domain model at all. Perhaps the access pattern choices prevent the use of a deep tree or graph of objects. Nonetheless we would like to access the all the data, whether in a domain object graph or not, in the most convenient way possible. This may be best accomplished using a *Domain Object Model Adapter*[2]:

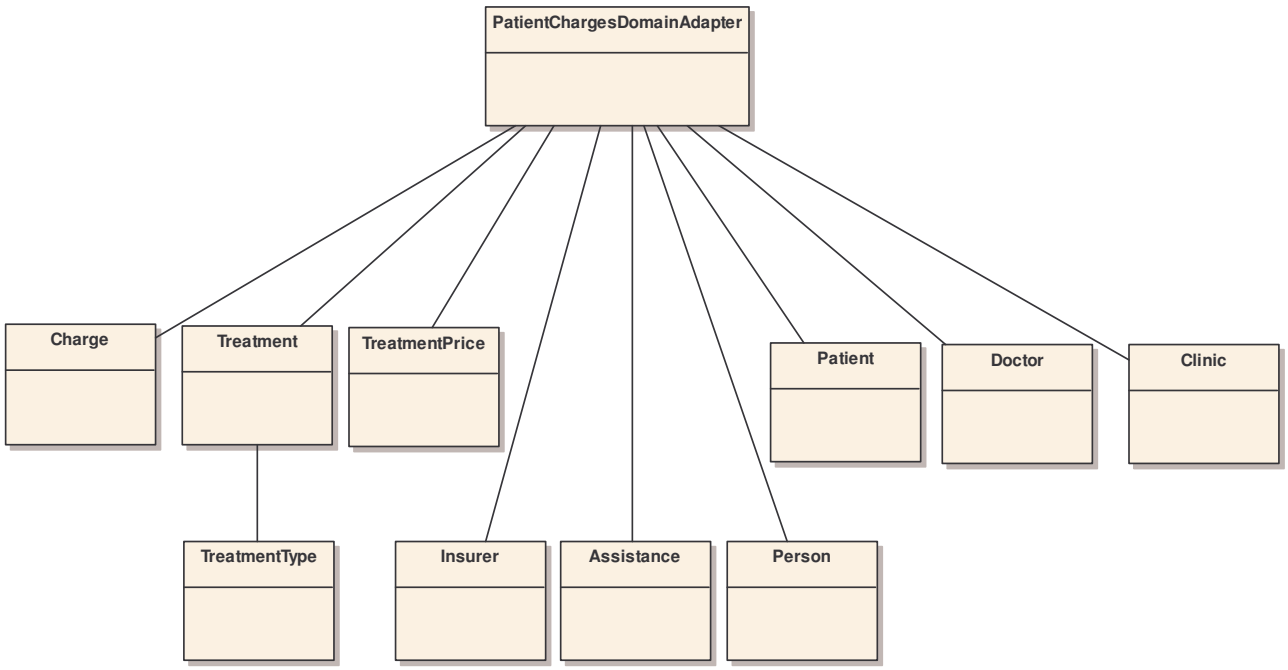
[1] See references: CoreJ2EE, P of EAA, and SDO.

[2] This pattern is not documented in this book. It is the basic *Adapter* pattern [GoF], but tuned to make data collected from a given data source appear to be domain objects, whether or not they are really domain objects managed by the *Domain Model* pattern [P of EAA].




The above is a graph that shows a domain-object view of the data we need to publish our business document. Some of the objects are navigable from multiple associated objects. For example, `Doctor`s may be found by looking in the `Clinic`(s) they work with, and by what `Patient`s the treat. Also `Patient`s may be found by the `Clinic` they are treated in or by the `Doctor`(s) treating them. But depending on the data access and object container patterns in use, your *physical* model may or may not at all look like this. However, in the above example it is class `PatientChargesDomainAdapter` that provides a *logical* and *optimal* view of the physical data.

This *Adapter* [GoF] should not be confused with the *Remote Façade* pattern [P of EAA]. It does not provide core business logic. It just simplifies access to the underlying data objects, giving them a unified and logical access point. In fact, you may want to tune the *Adapter* to provide an even simpler logical view of the physical data:




So what is the difference? The public methods on the first implementation of the `PatientChargesDomainAdapter` would look like this:

```
public class PatientChargesDomainAdapter {
  public static PatientChargesDomainAdapter getInstance(long aPatientId) . . .
  public Clinic getClinic() . . .
  public Patient getPatient() . . .
}
```

In order to get the `Patient`'s `Doctor`(s), we would need to do the following:

```
PatientChargesDomainAdapter pca = PatientChargesDomainAdapter.getInstance(id);
List doctorsList = pca.getPatient().getDoctors();
```

On the other hand, the second implementation of the *Adapter* would have a method for accessing each of the major *logical* domain objects, regardless of the natural navigation necessary to provide the access:

```
public class PatientChargesDomainAdapter {
  public static PatientChargesDomainAdapter getInstance(long aPatientId) . . .
  List getCharges() . . .
  Clinic getClinic() . . .
  List getDoctors() . . .
  Patient getPatient() . . .
  List getTreatments() . . .
  TreatmentPrice getTreatmentPrice(Treatment aTreatment)  . . .
}
```

In that case we would now be able to access the `Patient`'s `Doctor`(s) more easily:

```
PatientChargesDomainAdapter pca = PatientChargesDomainAdapter.getInstance(id);
List doctorsList = pca.getDoctors();
```

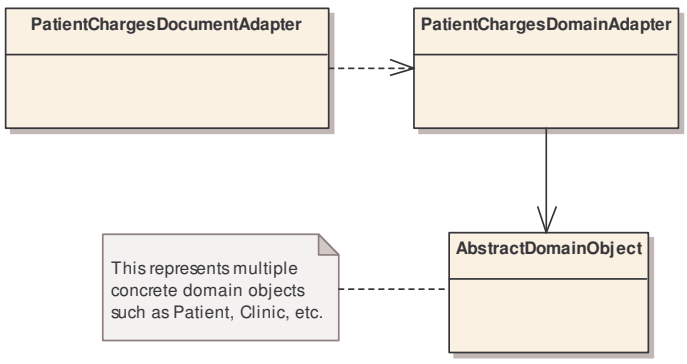
A better adapter is one that makes access to the publishable information convenient to the aggregation process. It does more adapting per the requirements of its client consumers.

That's about it for the information access portion of the data aggregation process. But it is not all there is to aggregating the data. The other half of the aggregation process is the one that gathers the data into a format that is more natural for the publication process. And not surprisingly this half of the aggregation process also uses an *Adapter* [GoF]. This particular *Adapter* is different from the *Domain Object Model Adapter* pattern. It is not domain-model centric. It is channel-document centric, and thus is called the *Channel Document Adapter*[3], as is shown in the introductory diagram.

This version focuses on adapting raw data into publishable document information. For example, the `Charge` domain object provided by the `PatientChargesDomainAdapter` contains information that indicates how much a `Patient` was changed for a given `Treatment`. The monetary value is stored in the denomination that the corporation deals in. But if the clinic in which the patient was treated is in a different country the charge must be provided in the local denomination using the exchange rate governed by the business. The conversion is provided by a service that has its own domain data, and is not part of the primary domain model in the `PatientChargesDomainAdapter`.

`PatientChargesDocumentAdapter` is a fitting name for this class. As suggested above, the data adaptation it performs focuses on isolating business logic. Another example of this class' business logic adaptation responsibilities is to perform any necessary calculations (or to delegate them to a rules engine) that are not stored in the data source, but that are displayed in the published document. It would also associate the description of the treatment found in the `TreatmentType` object with the code and number of units administered in each `Treatment` object.

A *Channel Document Adapter* provides accessor methods, such as `getPatientChargesInfo()`. Like methods provide a better set of associated data as far as the *document* is concerned than does the *Domain Object Model Adapter* classes' methods. Besides, running special business logic may produce the channel document associated data. But frankly, some of the data—perhaps even most of the data—may be a straight pass through from the underlying *Domain Object Model Adapter*. We now have the following class structure to our data aggregation process:




[3] Like the *Domain Object Model Adapter*, this one is also based on the standard *Adapter* pattern [GoF].

***Why Two Adapters?*** You may be strongly questioning why we should go to the trouble of providing two *Adapters*, especially when one may simply pass data directly through from the one below it. This is a separation of concerns decision. The lower-level *Adapter* deals with providing data access as a *domain model*. The higher-level *Adapter* is concerned with appropriately assembling to domain model data in a manner that is best for the *document* that will be hosting it.

Further, the *Domain Object Model Adapter* is reusable. If at some point in time you want use these classes for another purpose. You will be able to do so easily because your concerns are separated into two patterns and implementing classes.

Should your *Channel Document Adapter* subclass your *Domain Object Model Adapter* so you don't have to duplicate methods that simply pass data as-is up the chain? I say "no." If you subclass the *Domain Object Model Adapter* classes, besides showing through public methods that you want the client to see, it will also show through public methods that client of the *Channel Document Adapter* should not see. If such methods are visible to the client they may be used to access data in a way that the document is not prepared to host.

Now that the single-channel data aggregator is understood, the multi-channel aggregator is a cinch. All we need to do is provide a pair of *Adapters* for each channel being published: one for *domain model* concerns and one for the *document* concerns. The key is, what channels should you publish?

Obviously you will have to publish every channel that contains data that will be hosted by the end document. But I suggest that a better use of this pattern *always* publishes multiple channels. Why? I believe that this pattern should always use at least two channels, the primary data channel and a *content management* channel. This point will be become clearer as we advance to *Structured Data Stream Assembly* and *Page Layout and Content Formatting* processes.

## *Structured Data Stream Assembly*

At this point we have access to all the data objects that will ultimately serve as input to the published business document. However, the input has not yet been assembled into a sequence of useful structures that are suitable for input to the eventual *Page Layout and Content Formatting* process. That's what happens now.

What data format will work best for each particular publishing situation? Well, each publication is unique. So it is impossible to select a single format that works best for *every* publishing situation. Really, it would be best to be able to tune the data format for each and every scenario. But unless this is done carefully we may end up supporting a plethora of specialty formats. So what works?

I believe that structured data stream assembly is an excellent way to apply XML. While I openly admit that XML has become overused, publishing is actually one of the key motivations for its existence. Especially when coupled with the complementary transformation technologies (see below), we have a strong basis for using this standard document markup language.

If you don't believe that XML is the best way to create a structured document for your publishing process, consider the alternatives. Basically we have the additional

choice of using a proprietary format, or SGML, the more complex predecessor to XML. Given that our own proprietary structured document format may become open and standards-based simply by surrounding data with XML tags and attributes that are described by a DTD or XSD (Schema) is a very compelling approach.

Admittedly it is not necessarily the best approach for pattern authors to insist on implementation details. So for the remainder of the pattern I will use XML and related technologies in my examples, but you should ultimately use the principles presented here in conjunction with the structured data formatting approach that best suits your enterprise.

Structure data so that related content and attributes are together. In XML we use a hierarchy of logically nested elements. For example, if we want to describe a book in XML, at a high level it might look a lot like this:

```
<book>
  <info>
    <isbn/>
    <title/>
    <author/>
  </info>
  <toc/>
  <preface/>
  <chapters>
    <chapter number="1">...</chapter>
  </chapters>
  <index/>
</book>
```

The `book` itself is the outer structure. Inside this structure we find standard `info` such as the ISBN, the title, and the author. Each of these is represented by a dedicated element within `info`. Following this is the table of contents, or `toc`. Next a `chapters` element encapsulates a separate `chapter` element for each chapter. The last major substructure in the structured document is `index`. What is being emphasized here is that related data is clustered together in related structures and substructures.

It is likely that related data will be displayed in the published document within the same proximity. So it makes sense to collect it into the same area of the structured data stream. In fact, there is nothing wrong with setting up the structured document to follow the basic logical flow of the published document. It is not absolutely necessary to do so since the structured document is not trying to be the highly readable and printable published document, but it may help to expedite the development process to do so. Certainly, however, if the final published document order and layout change over time there is no reason to change the order of content in the structured document.

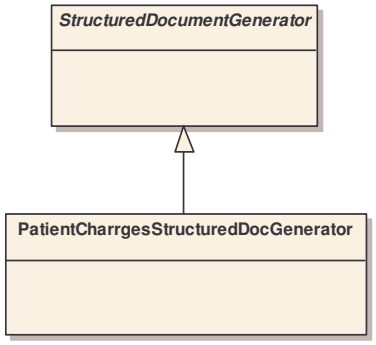
I want to emphasize that the *stream* part of the *Structured Data Stream Assembly* process should if all possible truly be a stream. That is, the structured data is written to an output stream, that becomes an input stream to the *Page Layout and Content Formatting* process. While text characters can be written to a disk file as a stream and read from the disk file through a stream, it is advantageous to use memory streams instead. Keeping the structured document entirely in memory has obvious performance benefits. It also reduces the amount of housekeeping necessary to clean up after the document is published. If your publisher is multithreaded the output data stream may be read as input by the *Page Layout and Content Formatting* process as the data streams.

There are situations in which the use of disk files is a necessity. One situation has to do with the size of the documents being generated. Another has to do with the total number of output parts. It may be that several structured documents must be generated for the publication of a single finished document. In this case, especially if the structured documents are large, it may be best to write output to disk files.

Anytime that the structured data stream must be placed into one or more disk files it is best to create them in a temporary location. Since the structured documents do not need to be accessible to consumers there is no need to make the temporary directory and files relative to the Web site. Rather any temporary directory will do. We must only exercise care to clean up after the end document is generated. It may be that some temporary files are used by more than one document. For example, document templates and standard boilerplate content could be used across many documents. In that case it is advantageous to preserve multi-use streams (or files) around for subsequent generation processes.

This leads us to the classes needed to assemble the structured data stream. In this pattern's introductory diagram note the abstract and concrete classes. The abstract class, `StructuredDocumentGenerator`, contains default behavior for the generation process. This includes the creation of memory and file streams, and their cleanup. If XML is used this class would also include default behavior for creating the structured documents, such as an interface to a standard XML generation API.

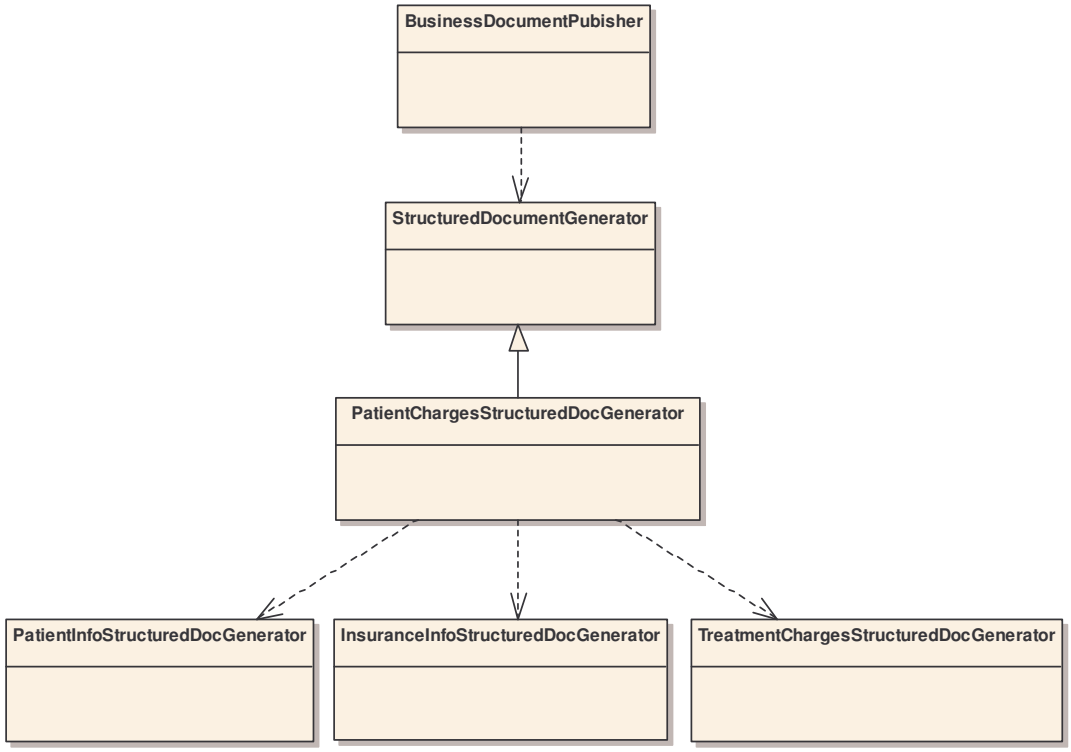
The `ConcreteStructuredDocumentGenerator` class represents the class or classes you would create specifically for the generation of the structured documents for a particular publication. It is not the actual name, however. The prefix name would be based on the publication at hand. Returning to the patient charges example, these are the possible classes involved:




There is no reason that the number of structured documents generated must be limited to one or even a few. Theoretically any number of structured documents could serve as input to the final business document. In that case the primary concrete structured document generator subclass becomes a driver for the creation of all such documents. It may make sense to generate multiple documents—perhaps even one per input data channel—under a few circumstances. For one, if creating a single structured document would make its structure so complex (e.g. many nested XML elements) that it would be more difficult to work with than multiple files, then use multiple files.

Another good reason is if you will be using some or most input data channels in multiple final business documents using the same data, it may be best to produce one structured document per input data channel. Doing so will make each concrete structured

document generator highly reusable. In that case you simply create a unique primary concrete structured document generator subclass to drive the operation of the lower worker-bee generators. Using the patient charges example we may formulate our structured document generators like the following:




All of the concrete structured document generators above are subclasses of the abstract class `StructuredDocumentGenerator`, although I do not show this detail here to keep the diagram simple. The class at the top of the diagram, `BusinessDocumentPublisher`, is the overall controller for the patient charges publishing processes. There may be no reason to provide a specialized publisher for each document type. The driver class `PatientChargesStructuredDocGenerator` controls all other worker-bee structured document generators. The controlling publisher need not directly know about driver structured document generator. In fact it is best if the publisher knows only about the `StructuredDocumentGenerator` abstract class. This will lend to this *Business Document Publisher* solution pattern serving as a pattern framework that is highly reusable.

But how does the `BusinessDocumentPublisher` know how to create each of its *Structured Document Generator* classes and their corresponding *Domain Object Model Adapters* and *Channel Document Adapters*? The short answer is, it doesn't know how to create them. The more complete answer is that object creation responsibility is placed on the `DocumentDescriptor`.

The `DocumentDescriptor` is a creational design-level pattern. It contains references to all the parts necessary to create the final business document. The client of class `BusinessDocumentPublisher` is responsible for setting up the

`DocumentDescriptor` before passing it to the publisher. The description of *Domain Object Model Adapter*, *Channel Document Adapter*, and *Structured Document Generator* may be done through configuration. For example a properties file or a more sophisticated runtime configuration management approach, such as a JMX (*Java Management Extensions*) MBean, could be used. Logically the descriptor is initialized as follows:

```
String patientId = patientAcct.getPatientId();
DocumentDescriptor docDesc = new PatientChargesDocumentDescriptor();
docDesc.setDocumentInputProvider(new PatientChargesDocInputProvider(patientId));
docDesc.addDomainAdapterClass(PatientChargesDomainAdapter.class);
docDesc.addChannelAdapterClass(PatientChargesChannelDataAdapter.class);
docDesc.addStructuredDocGeneratorClass(PatientChargesStructuredDocGenerator.class);
docDesc.addBusinessDocGeneratorClass(PatientChargesBusinessDocGenerator.class);
docDesc.setOutputDocumentType(DocumentDescriptor.PDF);
docDesc.setOutputDocumentFilename("PatientCharges.pdf");
docDesc.setOutputTempDirectory(env.getPublisherTempDir(patientId));
docDesc.addTargetPublisher(new EmailTargetPublisher(patientAcct));
```

The "add" methods facilitate the possibility of multiple participating classes. You should establish a specification for identifying the primary, controlling participant from each category, namely domain adapters, channel data adapters, structured document generators, and business document generators. For example, you could establish that the first class added to the `DocumentDescriptor` for each category serves as the primary controller.

Later inside the `BusinessDocumentPublisher` implementation the following creates the parts that the publisher uses to fulfill the publish request:

```
StructuredDocumentGenerator structDoc = docDesc.createStructuredDocumentGenerator();
BusinessDocumentGenerator bizDoc = docDesc.createBusinessDocumentGenerator();
. . .
```

Once the `StructuredDocumentGenerator` instance is created, all participating *Domain Object Model Adapters* and *Channel Document Adapters* are automatically created as well, and all dependencies are wired. Thus the data access and structured document workers are prepared for use. Next the publisher's specific `BusinessDocumentGenerator` is created. The various worker parts know what patient to generator a business document for by the `DocumentInputProvider` subclass established at the invocation of `setDocumentInputProvider()`.

## *Page Layout and Content Formatting*

There is certainly a lot of work put into preparing data as input to the publication of the final business document. But we have finally come to the point where the business document actually gets generated. As I stated earlier I am using XML for the structured document and related transformation technologies (XSLT) to create the finished document. Again, it is not absolutely necessary to use XML and XSLT. You could create your document using any proprietary page layout process. It's just that I think that the use of XML and XSLT will help your enterprise to remain open and for a broader cross-section of technologists to contribute to your development efforts.

Before going into the details of page layout, it is appropriate to establish the kind of document we will produce. Adobe's *Portable Document Format* (PDF) has earned worldwide acceptance as an open standard that supports a highly readable, highly

printable electronic document. I believe it would be foolish not to leverage the strengths of PDF and related tools and utilities.

These are the basic steps to producing a PDF using XML technologies:

1. Create an XSL style sheet for transforming the structured document
2. Parse the structured document(s) using a XSL and XSLT (transformation)
3. As your XSLT matches key elements from the structured document, output the structured document's content and attributes along with interspersed XML-based page layout and text formatting objects

The first two steps are probably intuitively obvious to you. If you are not familiar with XSL and XSLT, I suggest that you inform yourself about it. What may not be obvious is included in the third step; the generation of 'XML-based page formatting objects.' This is known as *XSL-FO*. The FO stands for *Formatting Objects*, and is the name of an open standard for creating high-precision documents such as PostScript and closely related PDF. Basically FO is a set of page layout and formatting commands in the form of XML tags that are interpreted by a document-formatting engine. As the document formatting engine processes the page layout and formatting commands, as well as text and image content, it uses a PostScript or PDF API to create respective documents.

There is actually no need to generate an intermediate set of structured documents that are passed through XSLT. We could generate the structured documents with FO commands embedded in them. However, this approach does not separate concerns—data from page layout—and is thus not reusable. I emphasize that the greater the potential for reuse the better the overall solution. Therefore, this pattern generates structured documents and then transforms the documents into a stream of content with page layout and content formatting commands using XML, XSLT, and FO.

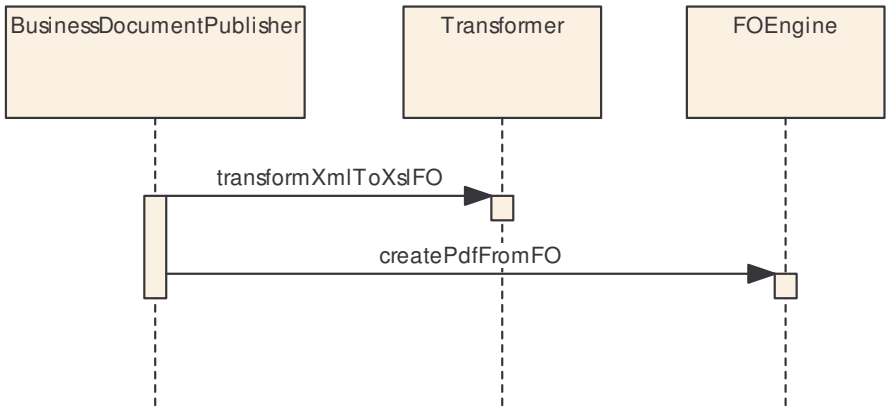
There are a few choices in XSL-FO engines available. One is the Apache Software Foundation's *FOP*, or Formatting Objects Processor. It is a fairly complete implementation of the standard. FOP has been known to exhibit some memory problems, but these are not uncommon to any and all FO processors, even commercial. More information on this is provided below under the *Robustness* subsection.

As you can see in the *Examples* section there are two kinds of FO commands. One is concerned with page layout, its dimensions. The other is for content formatting. Both are used in conjunction to make the generated business document look they way you want it to look.

Where does the XSL document come from? Recall that above I suggested using at least two data channels. One of the two minimum channels is the primary data channel—where your document's data comes from. The second of the two minimum channels is a *content management* channel. You would retrieve the XSL document from this special channel. Some content management systems can be slow to retrieve content *just in time*. Therefore, we must weigh the performance consequences of reading the XSL from the content management channel on demand. It may be best to cache often-used XSL in memory rather than reading it just in time.

After you have assembled all structured documents and XSL page layout and formatting templates into streams, it's time to run the transformation process. See the

*Examples* section for more detail, but the following diagram shows the high-level object interactions that run the end-to-end process:




Now that we have a highly readable, highly printable electronic document in our PDF, we are ready to publish the document to one or more of its final destinations.

## *Publishing*

Publishing is just a placeholder for a much more wordy description of the forth step in the overall pattern process. What publishing really means is putting the finished document in all the places it needs to go to be consumed.

If the business document is going to be referenced by pages on your business *Dynamic Web Site (page #)* then it will have to be placed in a directory that can be referenced by an HTML link in one or more Web pages. If the business document is going to be emailed to a consumer then it will have to be put into an email document as an attachment and sent. If the business document is going to be transferred to a partner, it will need to be passed on to a messaging or service dispatcher.

There are many possible targets for the finished business document. You might even need to support multiple targets for a single document. Whatever the case the `DocumentDescriptor`, defined in the above *Structured Data Stream Assembly* subsection, contains the targets used by the publisher to complete the process:

```
DocumentDescriptor docDesc = new PatientChargesDocumentDescriptor();
. . .
docDesc.addTargetPublisher(new EmailTargetPublisher(patientAcct));
docDesc.addTargetPublisher(new PayorPartnerTargetPublisher(partnerAcct));
```

This code supports two targets, an email publisher that sends to the patient directly and a partner publisher that ensures that the partner receives a copy of the patient charges statement. If the target were a *Dynamic Web Site (page #)* the `DocumentDescriptor` code would look something like this:

```
DocumentDescriptor docDesc = new CorporateReportDocumentDescriptor();
. . .
docDesc.addTargetPublisher(new WebTargetPublisher(corpReportUrl));
```

After the document is sent to all recipients then it is the publisher's responsibility to clean up. It does so by removing all temporary files and finished product business

documents from their temporary working directories. If all production and publishing is accomplished using memory streams, the garbage collection takes care of the clean up (assuming your implementation language supports it).

### *Robustness*

I repeat the robustness goals we have for our *Business Document Publisher*:

1. Publishing must be fault tolerant
2. The processing must scale as business needs grow
3. It must be executable on whatever enterprise platform and operating system that best supports the organization's environment

*Fault Tolerant*. This quality describes the enterprise's ability to recover automatically from processing crashes. With enough know-how and effort anyone can create a fault tolerant system. But this has already been accomplished by application server vendors through implementations of such standards as J2EE and .NET. It would be most productive to build your publisher behind such standard component technologies as EJB Session Beans, Web Services, messaging, and the like. These component frameworks implement the *Remote Façade* pattern [P of EAA] and facilitate advanced messaging patterns [EIP].

You can also take measures to prevent problems with the tools you use to process and publish. As previously stated, XSL-FO processors at times exhibit memory problems. These issues are generally related to the need for the processor to preserve page after page of formatted document in cache. This happens when the processor cannot resolve certain page elements until subsequent pages or even the entire document has been processed. One such situation is when a running footer includes page numbering in the form "Page X of Y." Y, or the total number of pages, is unknown until the document is fully processed. FOP will cache all pages in memory until it knows the total page count. It will then make a second pass through the document filling in the appropriate value of Y on each referencing page. As can be imagined, the larger the document the more memory needed to process it.

You have a couple of options to prevent such issues in your software. First, you may be able to avoid the document formatting constructs that cause the problems. Since this will not be possible in many cases, your second option is to protect your enterprise applications and systems from FO processing glitches and give them as much memory as possible to operate in. How?

Beware of using FOP or any other FO engine inside an application server container (such as a J2EE container) that also hosts your core application components. The container and your application probably already have serious memory overhead. Adding an FO engine with its page cache overhead can literally cause the container to encounter out-of-memory exceptions.

You should consider running your FO engine in its own dedicated application server instance or set of instances for increased robustness. All components used by your *Business Document Publisher* could be deployed in the same instance. Your core application components could interact with the *Business Document Publisher* via *Remote Façade* pattern [P of EAA] or via messaging [EIP].

You may also want to consider running your FO engine entirely out-of-band. In other words, you could run FOP as a detached process (Unix `fork()` with `exec()`, or Java's `Runtime.exec()` to achieve the same thing). This will allow you to get maximum memory usage out of FOP's JVM (for example).

*Scalability*. Not surprisingly most of the principals already stated regarding fault tolerance are applicable to scalability. Placing your *Business Document Publisher* and related components in a dedicated application server instance (or instances) will promote scalability as well. The application server instances (which may be a *Windows Server* in the case of .NET) may be on the same hardware as other instances to start out. Later as application throughput is more seriously challenged the *Business Document Publisher* could be moved off of a shared server and one to one or more dedicated hardware servers. Since the *Remote Façade* pattern [P of EAA] or messaging patterns [EIP] would already be in use, there would be little if any rework to scale the platform out to meet your growing business needs.

*Run Anywhere*. What this means is practicality—run where you *need* it to run. FOP, for example, runs on any platform for which a compatible *Java Runtime Environment* (JRE) is available. A Web Services (again via *Remote Façade*) or messaging interface would allow any external system to access the *Business Document Publisher* provided by FOP or another Java solution. There are also available XSL-FO engines available for the .NET platform, such as [get the product names]. The important thing to note is that XML, XSL, XSLT, and XSL-FO are open standards and, therefore, potentially have implementations for any platform.

## *Example*

My example focuses on the XSL-FO used to generate the business document. I do not provide a full implementation here, but the snippets are from a full example that I have posted on my Web site. See *Frameworks and Tools* below for more details. The examples here assume that the XML structured document has already been produced and that the XSL is now reading that document. You can find Java code snippets scattered thought this pattern that demonstrate the other important strategies it describes.

Here is a snippet of an XSL document that describes the page layout. This particular snippet defines the layout for only odd-numbered pages using Letter pages:

```
<xsl:template name="book-page-layout">
  <fo:layout-master-set>
    <fo:simple-page-master
          master-name="odd-book"
          page-height="11in"
          page-width="8.5in"
          margin-top="0.5in"
          margin-bottom="0.5in"
          margin-left="1.25in"
          margin-right="0.75in">
      <fo:region-body
            margin-left="0in"
            margin-right="0in"
            margin-top="0.5in"
            margin-bottom="0.5in"/>
      <fo:region-before
            region-name="op-before"
            extent="0.5in"/>
      <fo:region-after
            region-name="op-after"
```

```
            extent="0.5in"/>
      <fo:region-start
            region-name="op-start"
            extent="0in"/>
      <fo:region-end
            region-name="op-end"
            extent="0in"/>
    </fo:simple-page-master>
    . . .
```

Note that the `margin-left` attribute of the `fo:simple-page-master` element uses the value of 1.25 inches and the `right-margin` attribute uses 0.75 inches. This means that the surrounding page margins are both 0.75 inches, but that a gutter margin of an additional 0.5 inches is included at the left of the page. The pages will be bound on the left-hand side of odd-numbered pages, so the extra 0.5 inches of "gutter" or non-usable margin. We would naturally expect for even-numbered pages to specify the opposite layout since they will be bound on their right-hand side:

```
    <fo:simple-page-master
          master-name="even-book"
          page-height="11in"
          page-width="8.5in"
          margin-top="0.5in"
          margin-bottom="0.5in"
          margin-left="0.75in"
          margin-right="1.25in">
    . . .
```

The formatting commands are a bit different. While the page layout commands deal with the general size and shape of the page—its dimensions—the formatting commands make the content look the way you want it to look. Here is the opening matching and formatting commands of a document:

```
<xsl:template match="bookInfo">
  <fo:block xsl:use-attribute-sets="title-font-attr"
            text-align="center"
            space-after="8pt"
            margin-left="1.6in"
            margin-right="1.6in">
    Book, Author, and Schedule Information for Hosting Review on TheServerSide.com
  </fo:block>
```

The structured document takes the following form:

```
<bookSchedule>
  <overview/>
  <bookInfo>
    <title>Patterns of Enterprise Business Solutions</title>
    <description/>

    <author>
      <name>Vaughn Vernon</name>
      <contact type="email">vaughn@jubatus.com</contact>
      <about/>
    </author>
    <pages>450</pages>
  </bookInfo>
  <phases>
    <phase name="Phase 1">
      <description/>
      <deliverables>
        <deliverable section="1" chapter="1" title="Introduction" dueDate="8/9/04"/>
        ...
```

```
        <deliverable section="2" chapter="5" title="Dynamic Web Site" dueDate="9/6/04"/>
        ...
      </deliverables>
    </phase>
    ...
  </phases>
</bookSchedule>
```

When the XSL matches the `bookInfo` element it generates the corresponding `fo:block` element into its output stream. The document title text "Book, Author, and Schedule Information for Hosting Review on TheServerSide.com" is generated using `title-font-attr`. Above in the same template the `title-font-attr` global attribute is defined:

```
<xsl:attribute-set name="title-font-attr">
  <xsl:attribute name="font-family">Times-Roman, ZapfDingbats</xsl:attribute>
  <xsl:attribute name="font-size">12pt</xsl:attribute>
  <xsl:attribute name="font-weight">bold</xsl:attribute>
  <xsl:attribute name="text-align">justify</xsl:attribute>
</xsl:attribute-set>
```

When the `title-font-attr` is referenced in the block as an attribute set, the font definition is generated into the output stream. Besides 12 point Times-Roman Bold font, the `fo:block` also that the title text should be centered and that a blank line of 8 points in height should follow the title. I suggest that you see the full example on my Web site for more information on how XSL-FO works.

If you are creating PDF from XML using FOP, you may use a command-line invocation of the formatting engine, or you may run the engine using Java code to drive the processing. Here a command-line example:

```
fop -xml bookSchedule.xml -xsl bookSchedule.xsl -pdf bookSchedule.pdf
```

There are two steps to FOP's processing of this command. It first runs the XML structured document `bookSchedule.xml` through XSLT to produce the intermediate XSL-FO document. In the background this produces a stream that might be named `bookSchedule.fo`. FOP then processes the XSL-FO stream through the FO engine and produces its output, `bookSchedule.pdf`. The XML to XSL-FO transformation is not part of FOP's core processing responsibilities. In fact FOP simply delegates this transformation to the Apache *Xalan* XSLT engine. In effect, this is the Xalan command that could be issued and the following FOP command to accomplish the same thing as the preceding example command line:

```
xalan -in bookSchedule.xml -xsl bookSchedule.xsl -out bookSchedule.fo
fop -fo bookSchedule.fo -pdf bookSchedule.pdf
```

Running such commands may be a reasonable approach if you are running FOP out of band. It will allow Xalan and FOP to use the own memory space in a private JVM. If you conclude that this is not optimal for your enterprise you may always used the fine-grained control of using the Xalan and FOP Java APIs directly.

## *Consequences*

You will find these functional tradeoffs within the *Business Document Publisher* pattern.

- ***Open Versus Proprietary Processing***: Your enterprise may seem to lend itself to the use of proprietary implementations. This may be justifiable if you already have many legacy processes in place that were built long before open standards emerged. However, should your business continue to invest in the use of proprietary tools and formats as time passes? It may be best to begin a regimen of using open standards for all new business document publishing. Once you have gained expertise in the open technologies you may then determine that migrating proprietary legacy publishing processes to the open standard. This would have highest benefits if there is a lot of maintenance associated with the proprietary solution.
- ***Use of Open Source Or Commercial Products***: Since there are many open source products available for this pattern, it makes sense to evaluate them for your enterprise. Admittedly XSL-FO is less popular than general XML and XSL tools. Thus, we may find less support for FOP than for Xerces and Xalan. This fact may justify the evaluation of commercial XSL-FO processors. Note those available in the *Frameworks and Tools* section below.

## *Frameworks and Tools*

- **Xerces**: This is an Apache open source project and is an XML parser and document generation tool. It is a fine product, well supported, and widely used even by commercial products. To download see http://www.apache.org/.
- **Xalan**: This is an Apache open source project and is an XSL transformation engine. Xalan is also a fine product, it is well supported, and widely used even by commercial products. To download and learn more, see http://www.apache.org/.
- **FOP**: This tool is an Apache open source project that supports the processing of XSL-FO documents into several target formats. Among the most popular of support formats is PDF. Others include Java *AWT*-based screen output, MIF, PCL, PostScript, text, SVG, and direct to printer. See http://www.apache.org/.
- **Cacoon**: This is a full publishing framework provided by Apache. It includes a transformation engine and the FOP toolkit for publishing PDF and other XSL-FO output. Cacoon is traditionally recognized as a Web page publisher, and is therefore applicable to the *Stylized Page (page #)* pattern.
- **.NET Tools:** [Provide list.]
- **Resources:** Thanks go to my colleague *Glenn Pearson* for providing the XSLT and XSL-FO examples found in this book. Glenn has many years of SGML, XML, XSL, and XSL-FO experience. I have worked with Glenn on two different projects that made use of XSL-FO. I have posted Glenn's examples on my Web site, http://www.jubatus.com**.** Glenn can be reached at gpearson@documite.com.