Brian Loesgen
Charles Young
Jan Eliasen
Scott Colestock
Anush Kumar
Jon Flanders

BizTalk® Server 2010

UNLEASHED

SAMS

Brian Loesgen
Charles Young
Jan Eliasen
Scott Colestock
Anush Kumar
Jon Flanders

Microsoft® BizTalk® Server 2010

UNLEASHED

SAMS | 800 East 96th Street, Indianapolis, Indiana 46240 USA

Microsoft® BizTalk® Server 2010 Unleashed

ISBN-13: 978-0-672-33118-3

ISBN-10: 0-672-33118-7

Library of Congress Cataloging-in-Publication data is on file

Printed in the United States of America

First Printing September 2011

Trademarks

Warning and Disclaimer

Bulk Sales

Editor-in-Chief
Greg Wiegand

Executive Editor
Neil Rowe

Development Editor
Mark Renfrow

Managing Editor
Kristy Hart

Project Editor
Andy Beaster

Copy Editor
Keith Cline

Indexer
Lisa Stumpf

Proofreader
Apostrophe Editing Services

Technical Editor
Gijsbert in 't Veld

Publishing Coordinator
Cindy Teeters

Book Designer
Gary Adair

Compositor
Gloria Schurick

Contents at a Glance

Table of Contents

About the Authors

Brian Loesgen is a Principal Architect Evangelist with Microsoft on the Azure ISV team. Based in San Diego, Brian is a six-time Microsoft MVP and has extensive experience in building sophisticated enterprise, ESB, and SOA solutions. Brian was a key architect/developer of the "Microsoft ESB Guidance," initially released by Microsoft October 2006. He is a coauthor of the *SOA Manifesto* and is a coauthor of eight books, including *SOA with .NET and Windows Azure*, and is the lead author of *BizTalk Server 2010 Unleashed*. He has written technical white papers for Intel, Microsoft, and others. Brian has spoken at numerous major technical conferences worldwide. Brian is a cofounder and past-president of the International .NET Association (ineta.org), and past-president of the San Diego .NET user group, where he continues to lead the Architecture SIG, and is a member of the editorial board for the *.NET Developer's Journal*. Brian has been blogging since 2003 at http://blog.BrianLoesgen.com, and you can find him on Twitter as @BrianLoesgen.

Charles Young, MVP, MCPD, is a principal consultant at Solidsoft, an independent integration specialist working with BizTalk Server and related technologies. He has been a professional developer for a quarter of a century, worked for several years as a technical trainer, and has more than a decade of experience as a consultant. Charles has worked extensively with BizTalk Server since joining Solidsoft in 2003. He architects, designs, and implements enterprise-level integration applications for public- and private-sector customers, delivers seminars and workshops, and maintains a blog site. In recent years he has specialized in the area of decision systems and business rule processing and is vice-chair of Rules Fest, an annual technical conference for developers and researchers involved in the implementation of reasoning systems.

Jan Eliasen, MVP, MCTS, has a Master of Science degree in Computer Science and has been in the IT industry since 2003, currently working at Logica as an IT architect, focusing on delivering solutions to customers that meet the customers' needs. He started working with BizTalk 2002 just after graduation in 2003 and has been working with BizTalk ever since. He has passed the exams in BizTalk 2000, 2004, 2006, 2006R2, and 2010 and is a five-time MVP in BizTalk Server. He is a well-known contributor on the online MSDN forums and a blogger at http://blogs.eliasen.dk/technical/. You can follow him on Twitter as @jan_eliasen.

Scott Colestock lives and works in Minnesota. He has consulted on BizTalk, WCF, CQRS architecture, Agile methods, and general performance engineering. Recently, he has focused deeply on mobile and SaaS architectures using Windows Azure. He is an MVP and frequent speaker at conference events.

Anush Kumar is the chief technology officer at S3Edge (www.s3edge.com), a software solutions company focused on Auto-ID technologies, which he helped cofound following a distinguished career at Microsoft that spanned closed to a decade of working on multiple incubations from concept to shipping. In his last avatar at Microsoft, Anush was BizTalk RFID's leading light from early incubation of the project to its recent productization efforts, and has been heavily involved in the design and architecture of the RFID product, with multiple patents to his name. His efforts have also resulted in the vibrant partner and customer ecosystem for the product, and he is a sought-after speaker and thought leader in this space.

Prior to RFID, Anush worked on the business rules engine for BizTalk Server 2004, technology that has been deployed by several enterprise customers to improve agility and increase efficiency of their business processes. In his spare time, Anush enjoys backpacking off the beaten track; volunteers for organizations focused on education; and is a huge fan of Malcolm Gladwell, Guy Kawasaki, cricket, cooking, bungee jumping, and of course, All Things RTVS™ (http://rtvs.wordpress.com), his blog that spans RFID, and more! Anush holds a Bachelor of Engineering degree in Computer Science from University of Madras and a Master degree in Engineering from Dartmouth College.

Jon Flanders is a member of the technical staff at MCW, where he focuses on connected systems technologies. Jon is most at home spelunking, trying to figure out how things work from the inside out. Jon is the author of *RESTful .NET* and *ASP Internals*, and was a coauthor of *Mastering Visual Studio.NET*. Jon's current major interest is helping people to understand the advantages of REST and how REST connects to products such as SharePoint 2010. You can read his blog at http://www.rest-ful.net/

Dedications

I would like to dedicate this to my family, and to all the friends I've made over the years in the BizTalk community. I also want to thank the members of the stellar author team for all the hard work and effort they put in to make this book happen, and the team at Sams Publishing for making it all possible in the first place.
—Brian Loesgen

To the four girls and one boy in my life who amaze me by their care and love for each other and for me.
—Charles Young

This book is dedicated to my loving and caring wife, Helle, and our two sons, Andreas and Emil. Thank you for all your support when I was writing this book.
—Jan Eliasen

Thank to my beautiful wife, Tracy, and our fantastic kids (Nathan, Grant, Grace, and Anna) for your patience during Saturday and late-night writing sessions.
—Scott Colestock

The book is dedicated to all the guys and gal at S3Edge: I salute the tireless dedication and passion that's made our little start-up so much more than a place to work. To Mom and Dad for putting up with me through all the years To my lovely wife, Melissa, thank you for always being there for me, darling, and letting me live my dream.... And finally to Ambuloo, your turn now, sis!
—Anush Kumar

Acknowledgments

My thanks to Johan Hedberg, a solution architect at Enfo Zystems, who helps run the fabulous BizTalk User Group in Sweden, and who thoroughly reviewed the rules processing content. His feedback was invaluable and much appreciated. My thanks, also, to Gijs in 't Veld at Covast who played such a crucial role in reviewing the book as a whole, to the team at Pearson, to my sister Dorothy who doesn't know what BizTalk Server is, but is first in line to buy a copy of the book, and to my colleagues at Solidsoft who always ensure I keep my feet firmly planted in the real world of EAI.

—Charles Young

First, I want to thank my wife, Helle, and my children, Andreas and Emil. Thank you, Helle, for allowing me to spend all that time writing this book, and thank you for forcing me to keep writing even when I was tired of writing. Andreas and Emil, I am sorry for all the time I could not spend with you. Thank you for putting up with me, though.

Second, a big thank-you to Microsoft MVP Randal van Splunteren, who did me a personal favor by reviewing my stuff so that it wasn't too bad when I handed it in for the official review. And getting to that, also a thank-you to Microsoft MVP Gijs in 't Veld for doing the official review. And from the reviewers to the publisher: Thanks to Brook Farling for looking me up to begin with; thanks to Neil Rowe for taking over from Brook; and a big thanks to all involved at Sams.

Third, a thanks must go to the other authors (Brian, Charles, Scott, Jon, and Anush) for joining this team and writing their stuff, and a thanks also goes to my boss, Michael Hermansen, and my employer, Logica, for allowing me to spend time on this project.

Finally, another very big thanks to my wife, Helle.

—Jan Eliasen

Though I was only responsible for the two chapters on RFID, this would not have been possible for a first-time author without a stellar support cast in the background, starting with Ram Venkatesh, my colleague at S3Edge and the primary inspiration behind nudging me down the "authoring" path. The RFID chapters would not have been possible without your selfless help and guidance. So, many thanks, my friend! A big thank-you to Clint Tennill from Xterprise, and Gijs for their time and effort to review and provide invaluable feedback. To Brian and the rest of the veteran authoring crew, thanks for the opportunity to be part of this; you guys totally rock! And finally to Mark, Andy, Neil, and the rest of the crew at Pearson, thanks for your tireless efforts in getting us to the finish line; you guys have been consummate professionals all through the process, and just great to work with.

—Anush

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or email address. I will carefully review your comments and share them with the author and editors who worked on the book.

Email: feedback@samspublishing.com

Mail: Neil Rowe
Executive Editor
Sams Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

Foreword

In 2010, we celebrated two significant milestones—it marked both the 10-year anniversary of BizTalk and the release of BizTalk Server 2010. Over the past decade, there have been seven releases of Microsoft's enterprise integration platform, and it's become the most broadly deployed integration middleware technology on the planet (with nearly 12,000 customers worldwide). A key reason for this impressive growth was due to the explosion of the industry use of web services over the past decade; BizTalk Server helped fill a critical need as our customers began to transition from the world of client-server systems to service-oriented architectures. BizTalk Server was commonly used by our customers to service-enable existing LOB systems and extend the value of existing IT assets into newer applications.

As we look forward to the next decade, it's clear that we're beginning a similar magnitude of platform shift as the industry moves toward cloud computing. Although many are still grappling with how to begin the journey to the cloud, it's a matter of when they move not if—the long-term economic benefits to move to a public cloud computing model are undeniable, providing both cost-savings and simultaneous benefits in terms of business agility and ability for innovation. However, for most customers this journey will be a measured one—moving to the public cloud on their own terms and timeframe, and occurring in parallel with the continuing need to evolve and support an existing portfolio of on-premises applications.

BizTalk Server will play a key role—again—in this next industry platform shift. Integration technologies can play a key role as the gateway to the cloud by future-proofing today's applications so that even as you move ahead to the next-generation cloud platforms you can still leverage the existing investments you've made over the years. BizTalk Server 2010 has built in the capability to easily and securely bridge your existing integration business logic with the world of Windows Azure (Microsoft's cloud OS)—which can accelerate hybrid on/off premises application scenarios that we believe are critical to adoption of cloud computing.

The importance and relevancy of integration for the decade ahead is now more important than ever before, and this book can help you start. In this book, you get a comprehensive overview of BizTalk Server 2010 and its latest and greatest new capabilities. The team of authors (Brian Loesgen, Jan Eliasen, Charles Young, Scott Colestock, Jon Flanders, Anush Kumar) collectively have a tremendous wealth of practical, hands-on experience from implementing real-world integration solutions, and this book can help you start—regardless of whether you are new to BizTalk Server, or if you're an experienced developer wanting to stay current on the latest new features.

Burley Kawasaki
Director of Product Management
Microsoft Corp.

CHAPTER 3
Maps

IN THIS CHAPTER

- The Mapper
- Functoids
- Advanced Maps
- Building Custom Functoids
- Debugging
- Testing

As mentioned several times in Chapter 2, "Schemas," schemas are really important in your BizTalk solution. This is partly because they serve as the contract between systems and are therefore useful for determining which system is faulty. And it is partly because updating them is potentially particularly laborious because so many other artifacts depend on them.

One of the artifacts that depends heavily on schemas is a map. A map is a transformation from one *Extensible Markup Language* (XML) document into another XML document, and it is used in three places:

- To transform incoming trading-partner-specific or internal-system-specific messages into an internal XML format. This is achieved by setting the map on a receive port or on the receive side of a two-way port.
- To transform outgoing internal XML format messages into the trading-partner-specific or internal-system-specific formats they need to receive. This is achieved by setting the map on a send port or on the send side of a two-way port.
- To perform transformations needed inside business processes that do not involve receiving or sending messages to and from trading partners or internal systems.

Maps are developed inside Visual Studio 2010 using the BizTalk Mapper tool. This tool has a developer friendly interface, which allows you to have a tree structure view of both input and output schemas used by the map. When viewing these two tree structures, you can then use either

direct links from nodes in the source schema to nodes in the destination schemas, or you can use functoids that perform some processing on its input and then generates output, which can be used either as input for other functoids or as a value that goes to a node in the destination schema.

Although maps are developed in a nice, user-friendly interface and stored as a BizTalk-specific XML format, they are compiled into *Extensible Stylesheet Language Transformations* (XSLT) when the Visual Studio 2010 project is compiled. In fact, you can even provide your own XSLT instead of using the Mapper if you are so inclined or if you need to do complex transformations that you cannot do with the Mapper. Only XSLT 1.0 is supported for this, though.

Incoming files are either arriving as *Extensible Markup Language* (XML) or converted into XML in the receive pipeline, which happens before the map on the receive port is executed. Also, on a send port, the map is performed before the send pipeline converts the outgoing XML into the format it should have when arriving at the destination. This makes it possible for the Mapper and the XSLT to work for all files BizTalk handles because the tree structure shown in the Mapper is a representation of how the XML will look like for the file and because the XSLT can only be performed and will always be performed on XML. This provides a nice and clean way of handling all files in BizTalk in the same way when it comes to transformations.

The Mapper

This section walks you through the main layout of the BizTalk Mapper and describes the main functions and features.

Developing a map is done inside Visual Studio 2010 just as with other BizTalk artifacts. Follow these steps to add a map to your project:

1. Right-click your project.
2. Choose **Add**, **New Item**.
3. Choose **Map** and provide a name for the map. This is illustrated in Figure 3.1.

Layout of Mapper

After adding the map to your project, it opens in the BizTalk Mapper tool, which is shown in Figure 3.2.

The Mapper consists of five parts:

- To the left a Toolbox contains functoids that you can use in your map. Functoids are explained in detail later. If the Toolbox is not present, you can enable it by choosing **View**, **Toolbox** or by pressing **Ctrl+Alt+X**.
- A Source Schema view, which displays a tree structure of the source schema for the map.

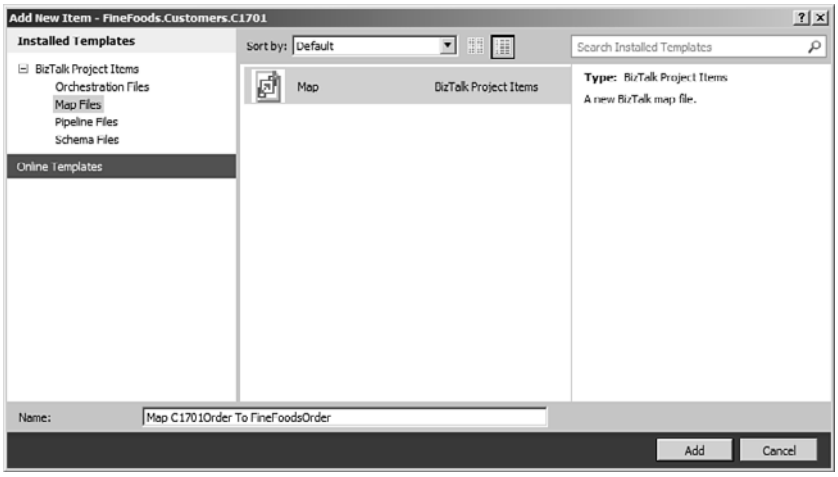

FIGURE 3.1 Add a new map to your project.

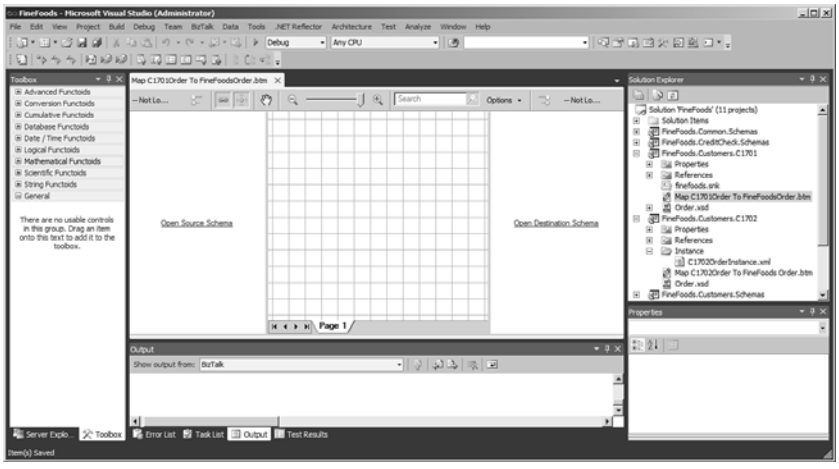

FIGURE 3.2 Overview of the BizTalk Mapper.

- The Mapper grid, which is where you place all functoids used by the map and also where lines between nodes in the source and destination schemas are shown. Above the Mapper grid there is a toolbar with some functionality that is described later.
- A Destination Schema view, which displays an inverted tree structure of the destination schema. An inverted tree structure means that it unfolds right to left rather than left to right, which is normal.

- The Properties window, which shows the properties that are available depending on what is the active part of the Mapper. For instance, it can show properties for a functoid in the map, a node in the source schema, or the map itself. If the Properties window is not present, you can always get to it by right-clicking the item for which you need the properties and choosing **Properties** or by clicking an item and pressing **F4**.

Initial Considerations

When developing a transformation, you usually assume that the input for the map is always valid given the schema for the source. This requires one of two things, however:

- Validation has been turned on in the receive pipeline, meaning that the pipeline used is either a custom pipeline with the XML validator component in it or validation has been enabled on the disassembler in use.
- You trust the sending system or trading partner to always send valid messages and therefore do not turn on validation. This can be done for performance reasons. The downside to this is, of course, that it can provide unpredictable results later on in the process and troubleshooting will be hard.

Either way, your business must decide what to do. Should validation be turned on so that errors are caught in the beginning of the process, or can it be turned off either because you trust the sender or because you decide to just deal with errors as they arise? As a developer of a transformation, you need to know the state of the incoming XML. If a map fails at some point, this can lead to unexpected behavior, like the following:

- Orchestrations can start failing and get suspended because the logic inside the orchestration is based on valid input.
- Incoming messages can get suspended if the map fails.
- If you validate your XML in a send pipeline and the map generated invalid XML according to the schema, the validation will fail, and the message will get suspended and not delivered to the recipient.

After this is dealt with, you can start looking at how to implement the map. Most of a map is usually straightforward, and you just specify which nodes in the source should be mapped to which nodes in the destination schema. For instance, the quantity on an order line is usually just mapped to the relevant quantity node in the destination schema that may have another name, namespace, or other. This works fine, as long as the cardinality and data type match between the source node and the destination node.

Special cases, however, must also be dealt with. Handling all the special cases can take a long time just to specify, and this time should be taken because you want to generate valid output. Determining how to handle these cases is usually not something a BizTalk developer can do alone because you need to specify what actions the business wants to perform in these cases. Therefore, this specification should be done in cooperation between a businessperson and a BizTalk developer. The most common special cases are described in the following paragraphs.

Different Data Types

If the source node and destination node have different data types, you might run into issues. Naturally, if you are mapping from one data type to another data type that has fewer restrictions, you are safe. If you are mapping form a node of type `decimal` to a node of type `string`, for example, you can just do the mapping because anything that can be in a node of type `decimal` can also be in a node of type `string`. The other way around, however, is not so easy. You have three options:

- Change the source schema either by changing the data type or by placing a restriction on the node that limits the possible values. You can use a regular expression to limit a string node to only contain numbers, for instance.
- Change the destination schema by changing the data type of the relevant node. Relaxing restrictions, however, can give you trouble later on in the business process.
- Handle the situation inside the map. After schemas are made and agreed upon with trading partners, they are not easy to change. So, you probably want to address this issue inside the map. You can use functoids, which are explained later, to deal with any inputs that are not numeric values.

Different Cardinality

If the source node is optional and the destination node is not, you have an issue. What you should do in case the input node is missing is a matter of discussion. Again, you have three options:

- Change the source schema by changing the optional node to be required.
- Change the destination schema by changing the relevant node to be optional.
- Handle the situation inside the map. You probably want to address this issue inside the map. You can use functoids to deal with the scenario where the source node is missing. This can either mean mapping a default value to the destination node or throwing an exception.

TIP

When developing a large solution that involves lots of trading partners, you will probably find yourself numerous times in the situation where someone says that they know the schema doesn't reflect it but some field that is optional is always present, so you can assume that in your map.

Well, don't do it! If the node is always present, the schema should reflect this.

If the schema doesn't reflect it, then in case the incoming XML doesn't have the node present (regardless of the promises that have been made to you), something unpredictable will go wrong. Besides that, the Mapper actually includes some extra logic in the generated XSLT in case of mapping optional elements, which can be avoided if the schema is properly developed.

Creating a Simple Map

To start creating the map, you must choose which schema to use as the input for the map and which schema to use for the output. These are also known as the source and the destination schemas of the map.

To choose the source schema, click **Open Source Schema** on the left side of the Mapper. Doing so opens a schema selector, as shown in Figure 3.3.

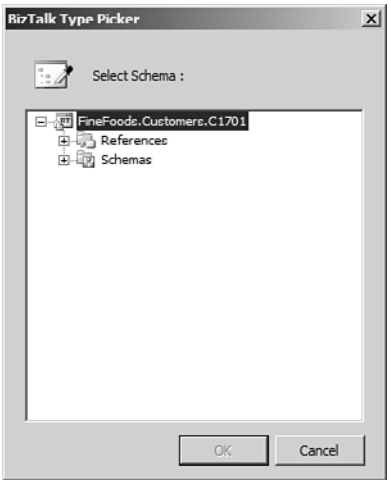

FIGURE 3.3 Choosing a schema to be used for the map.

In the schema selector, you can choose between schemas that exist in the current project or schemas that are in projects you have referenced from this project. You cannot add a reference from this window, so references must be added before choosing schemas in other projects. You choose a schema for the destinations schema by clicking **Open Destination Schema** and choosing a schema in the same way. If you choose a schema

that has multiple root nodes, you get a screen where you need to choose which one of them to use as the root node for the schema you have chosen.

TIP

If you use a schema from a referenced project as either the source or the destination schema, and this schema uses types defined in yet another project, the schema shows up wrong in the Mapper view. No errors are provided until you test the map with the validation options turned on. These options are described later. There isn't even a compile-time error or an error when validating the map. So, when adding referenced to projects with schemas, remember to check whether the schema you will be using itself uses types defined in schemas in other projects, and add that reference, as well.

NOTE

Designing a map by adding a new map to your project and choosing the source and destination schemas only allows for one schema as input and one schema as output. At times, it is desirable to have one schema as input and multiple schemas as outputs, thereby creating multiple outputs or to have multiple schemas as input and combining them into one schema or even have multiple inputs and multiple outputs. This can be achieved by defining the map inside an orchestration, and this is therefore covered in Chapter 4, "Orchestrations." These maps can be called only from within orchestrations and not from receive ports or end ports.

After choosing the two schemas involved, you are ready to start designing your map. This is mainly done by dragging links between the source schema and the destination schema and possibly doing some work on the source values before they are put into the destination schema.

For values that just need to be copied from a source node to a destination node, you can simply drag a link between the nodes in question. Just click either the source or the destination node and hold down the left mouse button while you move the mouse to the other element. Then release it. Doing so instructs the Mapper that you want the values from the source node copied to the node in the destination schema when the map is executed.

TIP

If the cursor turns into a circle with a line through it, this means that you cannot create this link. This can happen, for instance, if you want to map something into a field in the destination schema, which might not occur, like an element that has `maxOccurs` at `0`.

In between the source and destination schema is the Mapper grid. This grid is used to place functoids on, which perform some work on its input before its output is either used as input for another functoid or sent to the destination schema. Functoids are described later in this chapter. Figure 3.4 shows a simple map with a single line drawn and a single

functoid on it. The functoid is an "Uppercase" functoid that converts the input it gets into uppercase and outputs that.

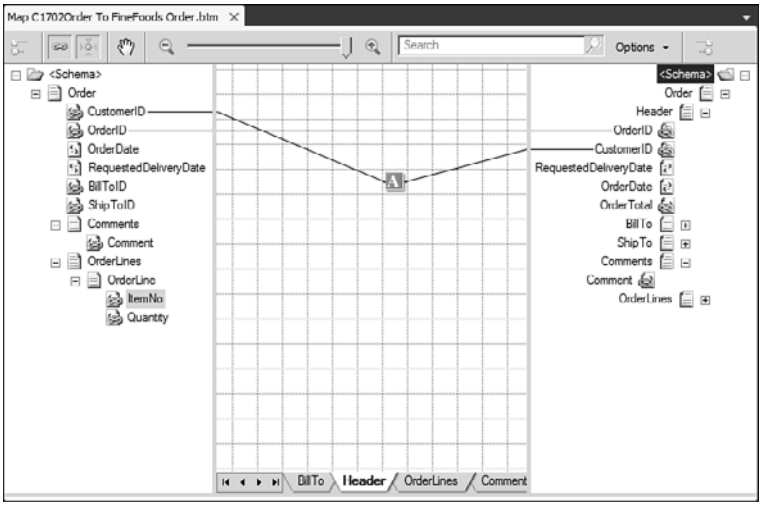

FIGURE 3.4 Simple map with one line drawn and one functoid used.

TIP

Be careful not to use too many functoids on the grid because this will seriously decrease readability. You can right-click the name of the map page (Default: Page 1) and choose **Add Page** to add another page. You can have up to 20 grid pages in your map. You can right-click a page name and choose **Rename Page** to rename it, and you can also choose **Reorder Pages** to change the order they appear in. Also, you can choose **Delete Page** if a page is no longer needed.

Pages are a great way of dividing your map into smaller, more manageable blocks. The functionality of the map is in no way influenced with how many pages you have and what functionality goes into what pages.

The map grid is actually larger than what you can see on your screen. If you move the mouse to the edge of the grid, the cursor changes to a large arrow, and you can then click to let the grid scroll so that you can see what is located in the direction the arrow points. You can also click the icon illustrating a hand in the toolbar at the top of the Mapper grid to get to the Panning mode, where you can click the grid and drag it around.

TIP

If you have so many functoids on one grid page that you need to scroll around to find the right place on the map grid, you can right-click the grid and choose **Grid Preview**. This will give you an opaque window providing easy access to choose which part of the map grid you want shown. After you've found the part of the grid you want to see in the map grid, just close down the opaque window.

If you need to move a functoid to another location on the grid, you need to first click it. When it is selected, you can drag it anywhere on the grid.

If you need to move a collection of functoids at the same time, you can click the grid and drag a rectangle to mark the functoids and links you want to move. After marking a rectangle on the grid, you can just click somewhere inside it and drag the entire rectangle to another location on the grid. Another option is to select multiple functoids/links by holding down **Ctrl** while clicking them. After they are selected, you can drag them to where you want them.

Sometimes you need to change one end of a link if, for instance, some destination node should have its value from another node than it does at the time. You can do this either by deleting the existing link and adding a new one or by clicking the link and then dragging one end of the link that has been changed to a small blue square to the new place. Changing the existing link instead of adding a new link has some advantages:

- All the properties you may have set on the link remain the same, so you do not have to set them again.
- If the link goes into a functoid, this will keep the order in which they are added. The order parameters are added to a functoid is important, so it is nice to not have to go in and change that order after deleting a link and adding a new one.

The window shown in Figure 3.4 has a toolbar at the top of the Mapper grid in the middle. This toolbar is new in the Mapper in BizTalk 2010 and contains some functionality that wasn't available in earlier versions of BizTalk.

One of the new features is the search bar. If you enter something in the search text box, the Mapper finds occurrences of this text within the map. The search feature can search in the source schema, the destination schema, and properties of the functoids such as name, label, comments, inputs, and scripts. You use the Options drop-down to the right of the search text box to enable and disable what the search will look at. Once a search is positive, you get three new buttons between the search text box and the Options drop-down. The three buttons enable you to find the next match going up, find the next match going down, or to clear the search. The search features are marked in Figure 3.5.

Another new option is the zoom feature. You get the option to zoom out, allowing you to locate the place on the grid you want to look at. For zooming, you can use the horizontal bar in the Mapper, as shown in Figure 3.6, or you can hold down the **Ctrl** key while using the scroll wheel on your mouse.

To let the map know that a value from one node in the source is to be mapped into a specific node in the destination schema, you drag a link between the two nodes. When you drag a link between two record nodes, you get a list of options:

- **Direct Link:** This creates a direct link between the two records. This helps the compiler know what levels of the source hierarchy correspond to what levels in the hierarchy of the destination schema.

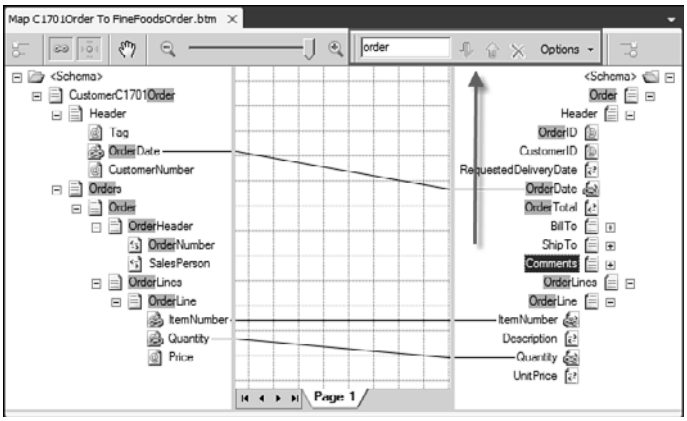

FIGURE 3.5 The search feature in a map.

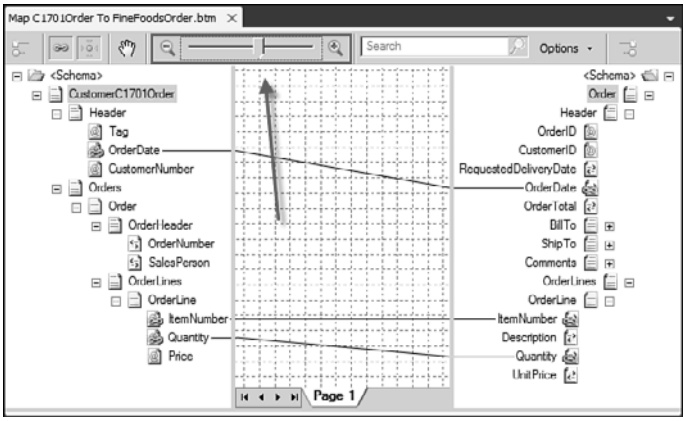

FIGURE 3.6 The zoom option in a map.

- **Link by Structure:** This lets the Mapper automatically create links between the child nodes of the two records you created the link between. The Mapper attempts to create the links based on the structure of the children.
- **Link by Name:** This lets the Mapper automatically create links between the child nodes of the two records you created the link between. The Mapper attempts to create the links based on the names of the children.
- **Mass Copy:** This adds a Mass Copy functoid that copies all subcontent of the record in the source to the record in the destination.
- **Cancel:** This cancels what you are doing.

This functionality is also new in BizTalk 2010. In earlier versions, there was a property on the map you could set before you dragged a link between two records.

Functoids and links can be moved between grid pages in two ways:

- After selecting one or more functoids/links, right-click them, and choose **Move to Page** or press **Ctrl+M Ctrl+M**. Doing so opens a small screen where you can choose between the existing pages or choose to create a new page to place the selected items on.
- Drag the selected items to the page tab of the page where you want them to appear. The page appears, and then you can place the items where you want them to be.

If you need a copy of a functoid, retaining all the properties of the functoid, you can also do this. Select a number of items and use the normal Windows shortcuts to copy, cut, and paste them. You can also right-click and choose **Copy**, **Cut**, or **Paste**. You can copy across grid pages, maps, and even maps in different instances of Visual Studio 2010. Some limitations apply to this, however, such as when links are copied and when not. For a full description, see refer to http://msdn.microsoft.com/en-us/library/ff629736(BTS.70).aspx.

For large schemas, it can be hard to keep track of which nodes are used in the map and in what context. To assist you, the Mapper has a feature called relevance tree view. This is a feature you can enable and disable on the source and destination schemas independently, and the feature is enabled or disabled using the highlighted button in Figure 3.7. As you can see, the relevance tree view is enabled for the destination schema and not for the source schema. The destination schema has some nodes coalesced to improve readability. This means that all the nodes in the `Header` record that are placed above the `OrderDate` node, which is the only node currently relevant for the map, are coalesced into one icon, showing that something is here but it is not relevant. You can click the icon to unfold the contents if you want. Records containing no nodes that are relevant for the map are not coalesced, but collapsed.

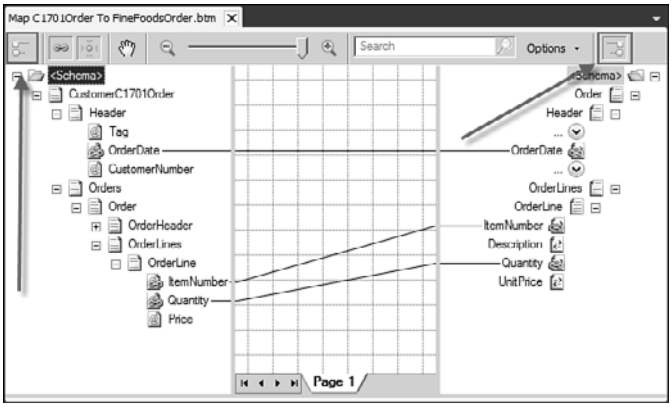

FIGURE 3.7 Relevance view.

If you have marked a field in the source schema and need to find the field in the destination schema to map it into, you can get some help from the Mapper, as well. This feature

is called Indicate Matches. If you select a node in the source schema, you can either press **Shift+Space** to enable it, or you can right-click it and choose **Indicate Matches**. Figure 3.8 shows how the screen looks after enabling the Indicate Matches feature on the `OrderDate` node in the source schema. As you can see, the Mapper adds some potential links to the map, and the one the Mapper thinks is the most likely is highlighted and thus the currently chosen one. If none of the suggestions match, you can press the **Escape** key or click with the mouse anywhere that is not one of the suggested links. If one of the links the Mapper suggests is the one you want, you have two ways of actually adding the link to the map:

- Use the mouse to click the link you want added to the map. Note that you cannot click the node in the destination that the link points to; it has to be the link itself.
- Use the up- and down-arrow keys to switch between the suggested links, and press the **Enter** key when the right link is chosen and highlighted.

If the feature guesses right the first time, you can add the link simply by pressing **Shift+Space** and then **Enter**. And you did not have to find the right place in the destination schema yourself.

Unfortunately, functoids are not part of this feature, so if you want the source node to be mapped into a functoid, this feature provides no help. You will have to do that yourself.

After a link has been dragged, it shows up in the Mapper as one of three types of links:

- **A solid line:** This is used for links where both ends of the link are visible in the current view of the Mapper, meaning that none of the two ends are scrolled out of the view.
- **A dashed line that is a little grayed out:** This is used for links where only one of the ends is visible in the current Mapper view and the other end is scrolled out of view.

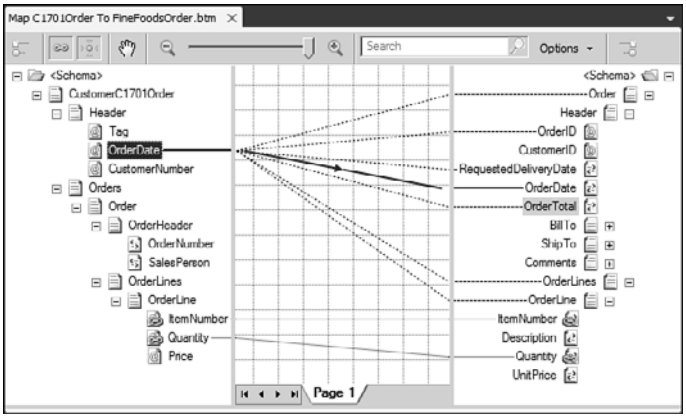

FIGURE 3.8 Illustration of the Indicate Matches feature.

- **A dotted line that is grayed out:** This is used for links where both ends are scrolled out of view but the link still goes through the current view of grid.

Figure 3.9 shows the different types of links.

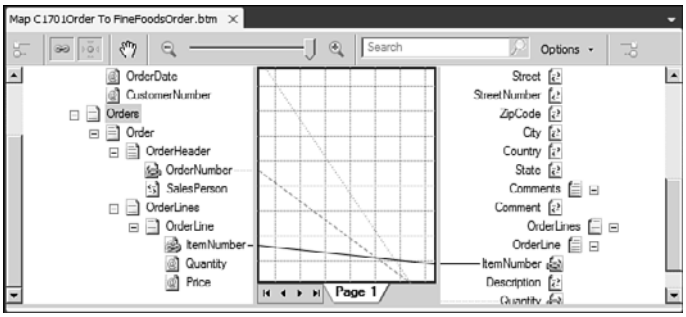

FIGURE 3.9 The three types of links in a map.

Because there may be a lot of links that are of the third type, where none of the ends of the link is visible, you might want to choose to not have these links shown at all. To do this, you can use a feature on the toolbar called Show All/Relevant Links. This is enabled using a button, as shown in Figure 3.10.

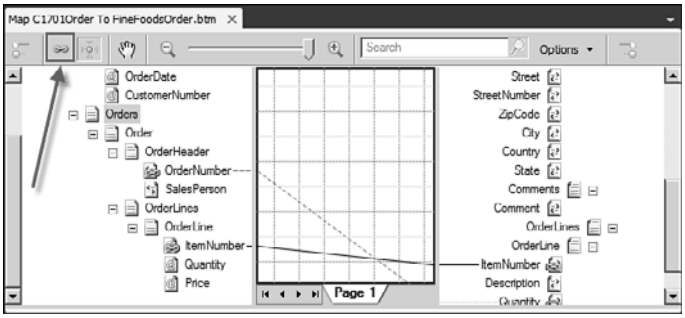

FIGURE 3.10 Feature to show all or relevant links.

As you can see from Figure 3.10, one of links that was also present in Figure 3.9 is no longer shown in the Mapper. The link still exists and is valid. If one or both of the ends of the link come into view, the link reappears on the grid.

When a map gets filled up with functoids and links, it can get hard to keep track of which links and functoids are connected. To help you with this, the Mapper automatically highlights relevant links and functoids for you, if you select a link, a functoid, or a node in either the source or destination schema. For instance, take a look at Figure 3.11.

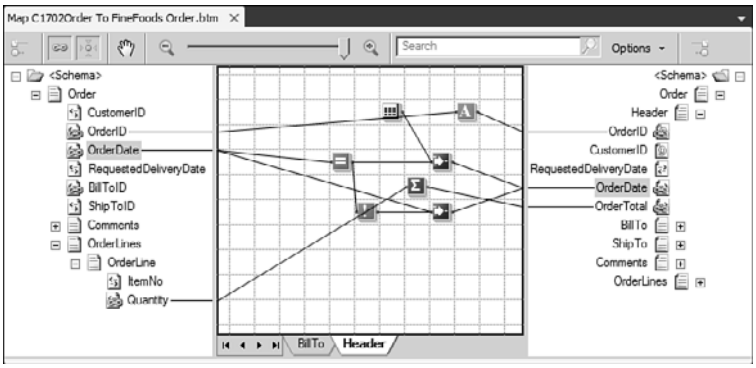

FIGURE 3.11 A map with lots of functoids.

Suppose you are troubleshooting to make sure the `OrderDate` in the destination schema is mapped correctly. If you click the `OrderDate` in the destination schema, you get the screen seen in Figure 3.12 instead. As you can see, the functoids and links that are relevant for mapping data into the `OrderDate` element have been highlighted and the rest of the links and functoids are now opaque, allowing you to focus on what is important. Had you clicked the link between the Equal functoid and the Value Mapping functoid, a subset of the links highlighted in Figure 3.12 would have been highlighted. If there are relevant links or functoids on another map page than the one currently shown, this is indicated by a small blue circle with an exclamation mark inside it to the left of the name of the page. Note also that the links have arrows on them, indicating the flow of data. This is also new in BizTalk 2010. In earlier versions of the Mapper, you could not have a functoid that gets its input from a functoid that was placed to the right of the first functoid on the grid. Now you can place your functoids where you want on the grid and the arrow will tell you which way the link goes. You cannot drag a link from a functoid to a functoid placed to the left of the first functoid, but after the link has been established, you can move the functoids around as you please.

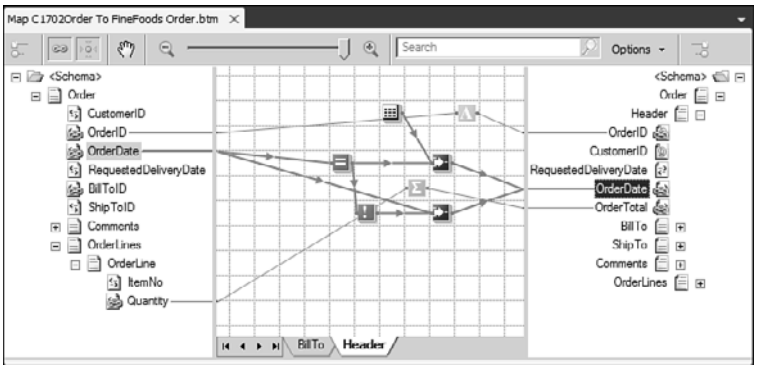

FIGURE 3.12 The links and functoids relevant for mapping into the **`OrderDate`** element.

Another feature is the Auto Scrolling feature. This feature, which is enabled and disabled using the button shown in Figure 3.13, allows the Mapper grid to autoscroll to find relevant functoids given something you have selected. If all the functoids had been out of sight in the Mapper grid and you then clicked the `OrderDate` in the destination schema with this feature enabled, the grid would autoscroll to the view shown in Figure 3.13. The Auto Scroll feature also applies to other parts of the map than clicking a node in a schema. If you click a functoid, for instance, the Mapper highlights relevant links and functoids that are connected to the selected functoid and uses the Auto Scroll feature to bring them into view, if enabled.

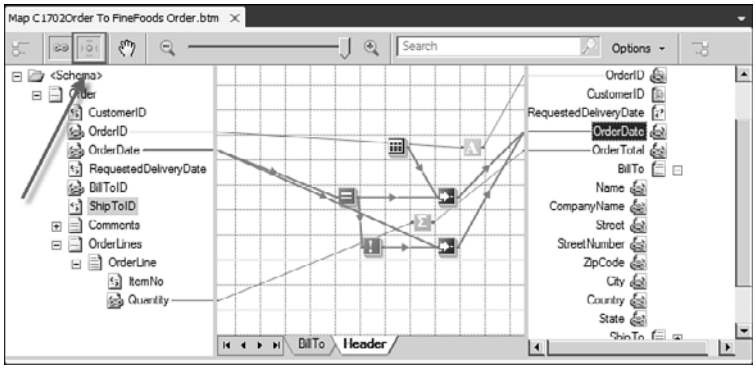

FIGURE 3.13 Example of using the Auto Scroll feature.

Sometimes you want insert a default value into a field in the destination schema. You can do this by clicking the field in question and in the properties for the field finding the property called Value and setting a value here. Other than setting a value, you can use the drop-down to select the <empty> option. This lets the Mapper create an empty field in the output. As explained in Chapter 4, it can be useful to have a way of setting values in messages outside a transformation. Also, empty elements are needed for property demotion, as explained in Chapter2, "Schemas."

If you choose to set a default value in a field in the destination schema in the map, you can no longer map any values to this node in the destination. If you open the *Extensible Markup Language Schema Definition* (XSD) and set a default value on the field in the schema itself instead of setting it in the Mapper, the Mapper uses this value, but you are allowed to map a value to the field, which overwrites the value set in the XSD. Unfortunately, there is no built-in support for using the default value from the XSD if the field that is mapped to a node is optional and not present in the source at runtime. You have to do this with an If-Then-Else sort of structure, as discussed later in this chapter.

If you choose to set a default value in a field in the source schema, this value is used only when generating instances for testing the map from within Visual Studio 2010. The value is not used at runtime when the map is deployed.

If you click the map grid, you can see the properties of the map in the Properties window. If this window is not present, you can right-click the grid and choose **Properties** or just click the grid and press **F4**. Table 3.1 describes the available properties for the map.

TABLE 3.1 Properties of the Map

Property Name	Description
General	
Ignore Namespaces for Links	Determines whether the map should be saved with information about the namespace of the nodes that are being linked.
Script Type Precedence	If a functoid that is used in the map can be both a referenced functoid or have (multiple) inline implementations, this property determines the order in which the implementations are to be preferred.
Source Schema	Read-only property that specifies the source schema to be used.
Target Schema	Read-only property that specifies the destination schema to be used.
Compiler	
Custom Extension XML	This property is used to point out a custom extension XML file that is used when providing your own XSLT instead of using the Mapper. This is explained more fully in the section "Custom XSLT."
Custom XSL Path	This property is used to point out a custom XSLT file that is used when providing your own XSLT instead of using the Mapper. This is explained more fully in the section "Custom XSLT."
Custom Header	
CDATA section elements	This property contains a whitespace-delimited list of element names that will have their values inside a CDATA construction to allow for otherwise-reserved characters. The whitespace can be a space or some other delimiter, but not, for instance, a comma.
Copy Processing Instructions (PIs)	Boolean value describing whether the map should copy any processing instructions from the source schema to the destination schema. Mostly used when transforming InfoPath documents.
Doctype public	Provides the value that will be written to the doctype-public attribute of the `xsl:output` element of the generated XSLT.
Doctype system	Provides the value that will be written to the doctype-system attribute of the `xsl:output` element of the generated XSLT.
Indent	Possible values are yes and no. If set to yes, the output of the map contains indentation to make the output more human-readable. Usually not needed, because systems often read the files and XML viewers show the XML nicely anyway.

TABLE 3.1 Properties of the Map

Property Name	Description
Media-Type	Used to specify the value of the media-type attribute of the `xsl:output` element in the generated XSLT. This determines the MIME type of the generated XML.
Method	Possible values are xml, html and text. The value specified goes into the method attribute of the `xsl:output` element in the generated XSLT.
Omit Xml Declaration	Can be true (default) or false. Is used as the value for the omit-xml-declaration attribute on the `xsl:output` element in the generated XSLT. Determines whether to omit the XML declaration at the top of the generated XML.
Standalone	Possible values are yes and no (default). Determines the value that is used in the standalone attribute of the `xsl:output` element in the generated XSLT.
Version	Specifies the value of the version attribute on the generated XML declaration, if any.
XSLT Encoding	This property contains a drop-down list of encodings. The chosen encoding is used to populate the encoding attribute of the `xsl:output` element. If you need another value than what is available, you can just enter it. The Mapper does not check the value, however, so check to make sure it is correct.

TIP

If you have a schema that has two elements on the same level of either the source or destination tree but with different namespaces, you need to set the **Ignore Namespaces for Links** to **False** because the Mapper cannot determine which of the nodes to use for the link when the map is opened up otherwise. The reason for the property to exist and not just default to "false" is that when the property is true, this allows you to change the namespace of your schemas without having to relink anything in the map. Also the .BTM file is easier to read when opened in a text editor like Notepad and it takes up less space.

NOTE

As explained later in this chapter, it is not possible to build a thread-safe referenced cumulative functoid. This means that if you are using any cumulative functoids, you should take care to have the Inline C# option or another appropriate inline option above the External Assembly option in the Script Type Precedence property.

NOTE

The CDATA Section Elements property is mapped directly to the `cdata-section-elements` attribute that can be present on the `xsl:output` element in XSLT and the use of this property can therefore be read about in an XSLT book. In short, any element with a name that appears in this list has its value wrapped inside a CDATA tag regardless of what level the element is on.

The only exceptions to this are elements with the same name, but in different namespaces, like elements that are of a type that comes from an imported schema. This also means that if you want those to have their content wrapped in a CDATA tag, you need to specify them as `ns0:elementname` in the CDATA Section Elements property. You can get he namespace prefix from the schema import dialog box where you imported the schema with the type you are referencing.

If you click a link in the map grid, you can see and change some properties of the link. If the Properties window is not present, you can right-click the link and choose **Properties** or click the link and press **F4**. Table 3.2 describes the properties for links.

TABLE 3.2 Properties for Links

Property Name	Description
General	
Label	In this property, you can specify a label that is used for this link.
Link Source	Read-only property that specifies the source of the link.
Link Target	Read-only property that specifies the target of the link.
Compiler	
Source Links	This property has three possible values: If you use the value Copy Text Value, the link copies the value from the source of the link to the destination of the link. If you choose Copy Name, the name of the source field is copied rather than its value. If you choose Copy Text and Sub Content Value, the source node and all its inner content is copied. This is like the `InnerText` property of a .NET `System.Xml.XmlNode`.
Target Links	This property determines the behavior of the link with regard to creating output nodes. Possible value are Flatten Links, Match Links Top Down, and Match Links Bottom Up. The value in this property allows the compiler to know how to match the source structure to the destination structure so that loops in the generated XSLT can be generated at the correct levels. Usually the default value is fine, but at times when you are mapping form recurring structures that are on different levels in the source and destination schemas, this can be useful.

Clicking the map file in Solution Explorer reveals some properties that you can set on the Visual Studio 2010 project item. Table 3.3 describes these properties.

TABLE 3.3 Properties for the Visual Studio 2010 Project Item

Property Name	Description
Advanced	
Build Action	This property determines what the compiler does with the .BTM file when building the project. A value of None instructs the compiler to ignore the file. A value of BtsCompile instructs the compiler to compile the map and include it in the assembly.
Fully Qualified Name	This is a read-only property displaying the fully qualified .NET name. It is a concatenation of the .NET namespace, a period, and the .NET type name.
Namespace	The .NET namespace this map belongs to. This has nothing to do with any namespaces used in schemas. As with normal code, the namespace is usually the same name as the project the file belongs to. Remember to change this if you move the .BTM file among projects.
Type Name	The .NET type name of the map. This corresponds to the class name of a class in a project. The type name is usually the same as the filename of the .BTM file.
Misc	
File Name	The filename of the map file. You can change this instead of renaming the file in Solution Explorer if you are so inclined.
Full Path	Read-only property containing the full path to the map file.
Test Map	
TestMap Input	Determines how Visual Studio 2010 can find a file to use as input when testing the map. A value of XML means that the file given in TestMap Input Instance is an XML file. A value of Native means that the file given in TestMap Input Instance is in the native format of the schema, which is in this case usually a flat file or an EDI schema, but it can also be XML. The last option is a value of Generated Instance, which lets Visual Studio 2010 generate an XML instance to be used for the test.
TestMap Input Instance	The full path to the file to be used for testing the map. This is used only if the TestMap Input property does not have a value of Generate Instance.
TestMap Output	Determines the format of the output from testing the map. Can be XML, which outputs the XML the map has produced, or Native, which translates the XML generated by the map into the destination format like EDI, flat file, or something else.

TABLE 3.3 Properties for the Visual Studio 2010 Project Item

Property Name	Description
TestMap Output Instance	The full path of where to place the output from testing the map. If not specified, the output is generated in your temp folder, and the output window provides a link to the specific file when testing the map.
Validate TestMap Input	Boolean value that determines whether the input for the map should be validated against the source schema before performing the map. If you know the instance to be valid, there is no need for validation, because this will only take time. On the other hand, making sure every time you test the map that the input is valid will confirm that you haven't accidentally changed the instance or the schema.
Validate TestMap Output	Boolean value that determines whether the output of the map should be validated against the destination schema. When developing your map, you will usually want to turn this off, to view the result of the partially complete map you have built. Then turn it on when you want to actually test the map in its entirety.

Functoids

BizTalk provides functoids to provide a way of performing some processing on the source values before it is copied to the destination. This processing can be anything from converting a string to uppercase over mathematical trigonometric functions to doing a database lookup.

The functoids are divided into categories, and the functoids in each category are described in the following sections. This chapter describes all the functoids. In the section "Advanced Maps," there are examples of advanced usage of the functoids.

After dragging a functoid to the grid, you can click it to see the properties of the functoid in the Properties window. If the Properties window is not present, you can right-click the functoid and choose **Properties**, or you can click the functoid and press **F4**. The Properties window contains the properties listed in Table 3.4.

TABLE 3.4 Functoid Properties

Property Name	Description
(Name)	The name of the functoid.
Comments	A text you can alter to contain some comments about this functoid.
Help	A textual description of the functoid that describes what the purpose of the functoid is.

TABLE 3.4 Functoid Properties

Property Name	Description
Input Parameters	Contains the text Configure Functoid Inputs. If you click in it, you get a button to the right of it with the ellipsis. If you click the ellipsis, you get to a window where you can add constant parameters to the functoid and change the order of the parameters. It usually also has text at the bottom that describes the parameters and what they are used for. After parameters are defined for the functoid, they are listed in this property rather than the text Configure Functoid Inputs.
Label	This is the only property you can edit. Here you can enter a friendly name for the functoid in question. This has no functional meaning, but it will perhaps ease the workload for a new developer taking over your map, and it can be used in utilities that do automatic documentation of your solutions.
Maximum Input Parameters	Describes what the maximum number of input parameters for this functoid is.
Minimum Input Parameters	Describes what the minimum number of input parameters for this functoid is.
Script	If the functoid is a Scripting functoid, which is explained later, you can click in this property and then click the ellipsis that appears to configure and edit the script the Scripting functoid should run when called.
Table Looping Grid	If the functoid is a Table Looping functoid, which is explained later, you can click this property and then the ellipsis that appears to configure the table looping grid.

TIP

Use labels on links extensively. When a link is an input to a functoid and you look at the functoid parameters, you can see something like Figure 3.14. If you label your links, however, the text in the label is shown rather than the default value, providing a much nicer interface to the parameter, enabling you to easily determine which parameter should come before which and so on. Figure 3.15 is the same as Figure 3.14, but with Comment from Source as the label on the link.

The screen for configuring the functoid parameters can be resized if necessary. Notice that when you drag a functoid to the grid, it has a yellow exclamation mark on it. This indicates, that the functoid is not fully configured, which at compile time would give you a warning or an error depending on the severity of the missing configuration. When you hover over a functoid that has the yellow exclamation mark on it, you get a tooltip describing what needs to be changed. Most of the time, you have missing input parameters or haven't connected the output of the functoid to anything.

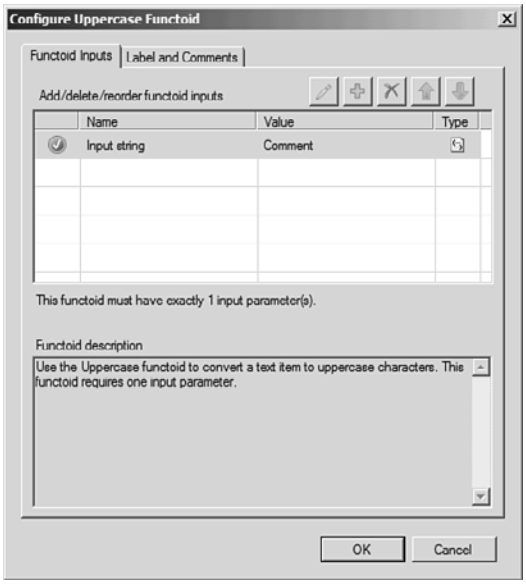

FIGURE 3.14 Parameters for a functoid without labels on the links.

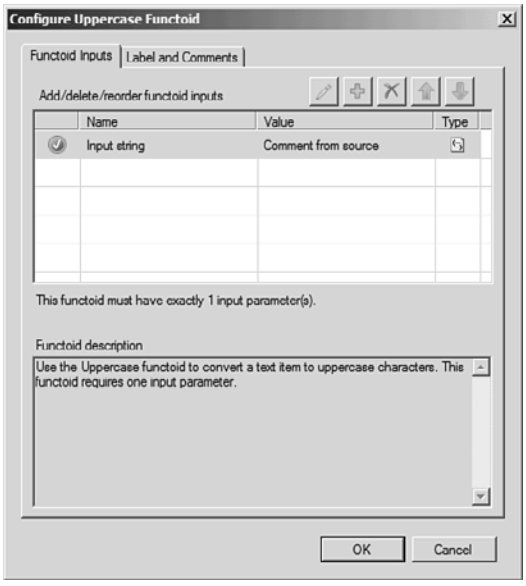

FIGURE 3.15 Parameters for a functoid, using labels on links.

String Functoids

The string functoids perform string manipulation on either an incoming value or a set of values. They all output a string, which can be used as the input for another functoid or sent directly to the destination schema. Table 3.5 lists the string functoids, their usage, and parameters.

TABLE 3.5 The String Functoids

Name	Description
Lowercase	This functoid takes in one parameter and returns the same string in lowercase.
Size	Returns an integer that equals the length of the input string. If the input string is null, a value of 0 is returned.
String Concatenate	This functoid takes in at least 1 string and a maximum of 100 strings. The strings are concatenated in the order they have been added as input to the functoid. The output is the concatenated string. Note this functoid cannot concatenate values in a reoccurring element in the source. You need a cumulative functoid for this, which is explained later.
String Extract	This functoid takes in three parameters and returns a substring of the first input. The second input determines the start position of the substring to be extracted and the third parameter determines the end position. Note that although the C# `String.Substring` method is actually used by the functoid, it wants the end position of the string to extract and not the length of the substring to extract. Also, contrary to the C# string methods, these parameters are 1 based and not 0 based, meaning that the first character of the input is a position 1.
String Find	This functoid takes in two parameters and returns the first found instance of the second parameter inside the first parameter. If the first parameter is `BizTalk` and the second parameter is `Talk`, the functoid returns the value 4. Note that this is also 1 based and not 0 based as is normally the case in C#.
String Left	This functoid takes in two parameters. The first is a string, and the second must be an int. The functoid returns the number of characters specified in the second parameter from the left of the string. If the second parameter is greater than the length of the string, the input string is returned.
String Left Trim	This functoid takes in a string and trims it from the left, effectively removing all characters from the beginning of the string until it reaches a nonwhitespace character. The characters that are defined as whitespace are the ones defined in C# as whitespace. These include spaces, carriage returns, line feeds, and others. Given null as input, the functoid returns the empty string.
String Right	This functoid does the same as the String Left functoid, only from the right rather than the left.

TABLE 3.5 The String Functoids

Name	Description
String Right Trim	This functoid takes in a string and trims it from the right. The functionality is the same as the String Trim Left functoid, only from the right rather than the left.
Uppercase	This functoid takes in one parameter and returns the same string in uppercase.

Mathematical Functoids

The mathematical functoids perform mathematical operations on their input and return an output that can be used as the input for another functoid or sent directly to the destination schema. Because the input parameters to a mathematical functoid are strings and a mathematical functoid returns a string, there are some methods used by the functoids to convert a string into a number to perform mathematical functions on the inputs. As a general rule, an input to a mathematical functoid that is not a number is ignored and if the parameter cannot be ignored, like either parameter in a division, an empty string is returned. Table 3.6 lists the mathematical functoids, their usage, and parameters.

TABLE 3.6 The Mathematical Functoids

Name	Description
Absolute Value	This functoid takes in one value and it returns the absolute value of the value. For a positive number, the absolute value is the same number. For a negative number, the absolute value is the number multiplied by –1.
Addition	This functoid takes in a minimum of 2 parameters and a maximum of 100 parameters. It sums the parameters and returns the result.
Division	This functoid takes in two parameters and returns the first parameter divided by the second parameter. If either parameter is not a number or if the second parameter is `0`, the empty string is returned.
Integer	This functoid takes in one parameter and returns the integer part of the value, effectively removing the decimal point and all decimals. This differs from the Round functoid in that it will never round up (only down).
Maximum Value	This functoid takes in a minimum of 2 parameters and a maximum of 100 parameters. It returns the largest number found among the parameters.
Minimum Value	This functoid takes in a minimum of 2 parameters and a maximum of 100 parameters. It returns the smallest number found among the parameters.
Modulo	This functoid takes in two parameters. It returns the remainder from performing a division of the first parameter by the second parameter.
Multiplication	This functoid takes in a minimum of 2 parameters and a maximum of 100 parameters. It multiplies all the parameters and returns the result.

TABLE 3.6 The Mathematical Functoids

Name	Description
Round	This functoid takes one required parameter and one optional parameter. The first parameter is the value to round, and the second is the number of decimals to which the first parameter should be rounded off. If the second parameter is not present, the first parameter is rounded off to a whole number. Standard rounding rules apply.
Square Root	This functoid takes in one parameter and returns the square root of it.
Subtraction	This functoid takes in a minimum of 2 parameters and a maximum of 100 parameters. It subtracts all parameters that are not the first parameter from the first parameter and returns the result.

All the functoids return an empty value if a specified parameter is not a number. Optional parameters can be omitted, but if they are used, they must also be numbers or an empty string is returned.

Logical Functoids

The logical functoids perform logical operations on their input and returns a Boolean value that can be used later on as the input for another functoid or to instruct the compiler how to construct the generated XSLT. How the logical functoids aid the compiler in constructing the generated XSLT is discussed in the "Advanced Maps" section. Because the logical functoids are used for either input to other functoids or to instruct the compiler, you cannot get the output of a logical functoid into your destination schema. If you want to do so, you can add a C# scripting functoid (described later) with the code as shown in Listing 3.1 and place that in between the logical functoid and the destination node. Table 3.7 lists the logical functoids, their usage and parameters.

TABLE 3.7 The Logical Functoids

Name	Description
Equal	This functoid takes in two parameters and returns true if the parameters are equal to each other. If they are not equal, a value of false is returned.
Greater Than	This functoid takes in two parameters. It returns true if the first parameter is greater than the second parameter. Otherwise, it returns false.
Greater Than or Equal To	This functoid takes in two parameters. It returns true if the first parameter is greater than or equal to the second parameter. Otherwise, it returns false.
IsNil	This functoid takes in one parameter, which must be a node in the source schema and cannot be another functoid. The functoid returns true if the node in the source schema is set to nil, like this: `<myElement xsi:nil="true" />`

TABLE 3.7 The Logical Functoids

Name	Description
Less Than	This functoid takes in two parameters. It returns true if the first parameter is less than the second parameter. Otherwise, it returns false.
Less Than or Equal To	This functoid takes in two parameters. It returns true if the first parameter is less than or equal to the second parameter. Otherwise, it returns false.
Logical AND	This functoid takes in a minimum of 2 parameters and a maximum of 100 parameters. It returns the logical AND of all the inputs, meaning that if just one of the inputs is false, a value of false is returned. Otherwise, a value of true is returned.
Logical Date	This functoid takes in one parameter and returns a Boolean value depending on whether the input could be parsed as a .NET `DateTime`.
Logical Existence	This functoid takes in one parameter. If the parameter is another functoid, the input must be a Boolean, because the Logical Existence functoid doesn't do any conversion of the input into Boolean. The functoid just passes the value it receives on. If the input is a node in the source schema, however, the functoid returns true if the node is present in the source schema and false otherwise.
Logical NOT	This functoid takes in a Boolean value and returns the negation of it, meaning true becomes false and vice versa.
Logical Numeric	This functoid takes in one parameter and returns a Boolean value depending on whether the input could be parsed as a .NET double.
Logical OR	This functoid takes in a minimum of 2 parameters and a maximum of 100 parameters. It returns the logical OR of all the inputs, meaning that if just one of the inputs is true, a value of true is returned. Otherwise, a value of false is returned.
Logical String	This functoid takes in one parameter and returns true if the parameter is neither null nor the empty string. Otherwise, it returns false.
Not Equal	This functoid takes in two parameters and returns true if the parameters are not equal to each other. If the two values are equal, a value of false is returned.

LISTING 3.1 C# Code for Getting Output of a Logical Functoid into a Destination Node

```
public string LogicalToString(string logical)
{
       return logical;

}
```

All the functoids that need to take a Boolean as input need to parse the input to make sure it is in fact a Boolean and not some random string. The algorithm for this is as follows:

1. The input is compared to the string `true` in a non-case-sensitive manner. If this succeeds, a value of `true` is returned, and processing stops.
2. The input is compared to the string `false` in a non-case-sensitive manner. If this succeeds, a value of `false` is returned, and processing stops.
3. The string is trimmed to remove leading and trailing whitespace.
4. The input is compared to the string `true` in a non-case-sensitive manner. If this succeeds, a value of `true` is returned, and processing stops.
5. The input is compared to the string "false" in a non-case-sensitive manner. If this succeeds, a value of `false` is returned, and processing stops.
6. The input is converted into a number. If this fails, a value of `false` is returned.
7. The number is greater than 0, a value of `true` is returned. Otherwise, a value of `false` is returned.

This algorithm is used by the Logical OR, Logical NOT, and Logical AND functoids to validate the input.

The Not Equal, Equal, Less Than or Equal To, Less Than, Greater Than or Equal To, and Greater Than functoids can all compare numbers to numbers and strings to strings.

Date/Time Functoids

The date/time functoids are a collection of functoids that deal with dates/times. Their output can be used as the input for another functoid or sent directly to the destination schema. Table 3.8 lists the date/time functoids, their usage, and parameters.

TABLE 3.8 The Date/Time Functoids

Name	Description
Add Days	This functoid takes two parameters and returns a new date as a result of adding the number of days specified in the second parameter to the date specified in the first parameter.
Date	This functoid takes no parameters and returns the current date.
Date and Time	This functoid takes no parameters and returns the current date and time.
Time	This functoid takes no parameters and returns the current time.

Conversion Functoids

The conversion functoids perform conversions on their input and return output that can be used as the input for another functoid or sent directly to the destination schema. Table 3.9 lists the conversion functoids, their usage, and parameters.

TABLE 3.9 The Conversion Functoids

Name	Description
ASCII to Character	This functoid takes in one parameter. If the parameter is a number less than 0 or greater than 127, an empty string is returned. Otherwise, the number is converted into the character that has this number in the ASCII table. 65 becomes A, for instance.
Character to ASCII	This functoid takes in one parameter. If the parameter is null or the empty string, an empty string is returned. If not, the first character of the string is converted into the ASCII representation. A becomes 65, for instance.
Hexadecimal	This functoid takes in one parameter. If the parameter is not a number, the empty string is returned. If the parameter is a number, the integer part of the number is converted to a hexadecimal value. The input is assumed to be a decimal value.
Octal	This functoid takes in one parameter. If the parameter is not a number, the empty string is returned. If the parameter is a number, the integer part of the number is converted to an octal value. The input is assumed to be a decimal value.

Scientific Functoids

The scientific functoids perform scientific mathematical operations on their input and return an output that can be used as the input for another functoid or sent directly to the destination schema. Table 3.10 lists the scientific functoids, their usage, and parameters.

TABLE 3.10 The Scientific Functoids

Name	Description
10^n	This functoid takes in one parameter. The functoid returns 10 lifted to the power of the parameter. For instance, a value of 2 results in 100, and a value of 5 results in 100000.
Arc Tangent	This functoid takes in one parameter. The functoid returns the result of performing the arc tangent function on the parameter.

TABLE 3.10 The Scientific Functoids

Name	Description
Base-Specified Logarithm	This functoid takes in two parameters. If either parameters is less than or equal to 0, an empty string is returned. Also, if the second parameter is 1, an empty string is returned. The functoid returns the base-specified logarithm function using the second parameter as base and performing the function on the first parameter. For instance, parameters 100 and 10 result in the value 2, and parameters 64 and 2 result in the value 6.
Common Logarithm	This functoid takes in one parameter. If the parameter is less than or equal to 0, an empty string is returned. Otherwise, the functoid returns the 10-based logarithm of the parameter. For instance, a value of 100 results in 2, and a value of 10000 results in 4.
Cosine	This functoid takes in one parameter. If the parameter is a number, the cosine function is called with the parameter as input, and the result of that is returned.
Natural Exponential Function	This functoid takes in one parameter. The functoid returns ℮ (The base for the natural logarithm) lifted to the power of the parameter. For instance, a value of 2 results in 7.389..., and a value of 3 results in 20.085....
Natural Logarithm	This functoid takes in one parameter. The functoid returns the natural logarithm of the parameter. The natural logarithm is the logarithm that is based on ℮.
Sine	This functoid takes in one parameter. If the parameter is a number, the sine function is called with the parameter as input, and the result of that is returned.
Tangent	This functoid takes in one parameter. The functoid returns the result of performing the tangens function on the input.
X^Y	This functoid takes in two parameters. The functoid returns the first parameter raised to the power of the second parameter.

For all the functoids you get an empty string as the result in case the input parameters could not be parsed as a number.

All trigonometric functoids like Sine, Cosine, Tangent, and Arc Tangent assume the input is in radians. This means that if you want to use any of these functoids on a value of 180° you need to convert the 180° into π first.

Cumulative Functoids

The cumulative functoids perform operations on reoccurring nodes in the source schema and output one value based on all the occurrences of the input node. The output can be used as the input for another functoid or sent directly to the destination schema. Table 3.11 lists the cumulative functoids, their usage, and parameters.

TABLE 3.11 The Cumulative Functoids

Name	Description
Cumulative Average	This functoid takes in all the occurrences of the input and outputs the average of the input values.
Cumulative Concatenate	This functoid concatenates the occurrences of the input into one string, which is then output.
Cumulative Maximum	This functoid takes in all the occurrences of the input and outputs the largest number found among the input values.
Cumulative Minimum	This functoid takes in all the occurrences of the input and outputs the smallest number found among the input values.
Cumulative Sum	This functoid takes in all the occurrences of the input and outputs the sum of the input values.

All the cumulative functoids take in two parameters. The first parameter is the value to be accumulated, and the second is a scoping parameter. The scoping parameter is used to generate the XSLT. If no value is passed on, a default value of 1000 is used in the XSLT. If a value is actually passed into the parameter, this is used as an indication of what level of scope to do the accumulation. A value of 0 means that all occurrences of the first parameter are accumulated and the functoid will therefore have only one output. A value of 1 means that the values are accumulated for each parent of the first parameter and an output is therefore generated for each parent. A value of 2 means that the values are accumulated for each grandparent of the first parameter and so on. Note that it is not the functoid that contains this logic. The generated XSLT will do all this for you, making sure that the functoid is called for each level of scoping needed.

NOTE

As explained later in this chapter, it is not possible to build a thread-safe referenced cumulative functoid. Therefore, if you are using any of the built-in cumulative functoids, take care to have the Inline C# option or another appropriate inline option above the External Assembly option in the Script Type Precedence property of the map.

Database Functoids

The database functoids can be split into two categories; Those that do database lookup and those that do cross referencing.

Database

The database functoids are used for looking up values in an ADO.NET-compliant database that can be used as the input for another functoid or sent directly to the destination schema. Table 3.12 lists the database functoids, their usage, and parameters.

TABLE 3.12 The Database Functoids

Name	Description
Database Lookup	This functoid takes in four parameters, which are all required. The first parameter (p1) is a lookup value, which could be some ID from the source message. The second parameter is a connection string that instructs the runtime where to do the lookup. The third parameter (p3) is the name of the table to do the lookup. The fourth parameter (p4) is the name of the column that is used to find the lookup value. BizTalk creates a query like this: `SELECT * FROM p3 WHERE p4 = p1;` and connects to the database, and thus selects all values from the row in the table specified that has the right lookup value in the column specified in the fourth parameter. This requires the column in the fourth parameter to be a unique key in the table. If it is not a key, the first row is returned. This functoid cannot be connected to an output node, because it outputs an entire row. Use the Value Extractor functoid, which is explained later, to extract a column value from this row.
Error Return	This functoid takes in one parameter, which is a Database Lookup functoid. The functoid outputs any error message that has arrived from performing the query against the database. If no errors occur, the functoid has an empty string as output.
Value Extractor	This functoid takes two parameters, which are both required. The first is a Database Lookup functoid. The second is the name of the column from the output row from the Database Lookup to extract. This effectively pulls out one specific value from a named column in the result set of the database query, and this value can then be output to the destination schema.

NOTE

Hard-coding the connection string inside your functoid is probably not what you want to do because it requires you to change it manually and recompile for each environment your map is deployed to. Rather, you should keep the connection string in some configuration storage and retrieve it at runtime. Options for this include *Single Sign-On* (SSO), a configuration file, a database, a .NET assembly, and others.

The preceding options are generic for all configuration options you might use throughout your solution, but one last option that is specific for the database lookup functoid exists. You can enter `File Name=<PathToUDLFile>` as the connection string. This requires, of course, that the file can be read at runtime by the host instance that is running. Parsing a UDL file is slow, so the options listed previously are recommended considering performance.

Cross Referencing

The cross-referencing functoids are used for looking up values in a database and using the values retrieved as the input for another functoid or sending it directly to the destination schema. This is used to map values that are specific to one system to the corresponding values from another system. For instance, you can use this setup to find your *enterprise*

resource planning (ERP) system item number based on the item number your customer has supplied you with in the order you have received.

The cross-referencing functoids are restricted to and make use of the 9 xref_* tables that are present in the BizTalkMgmtDb database. Importing data into these tables is done via the BizTalk Server Cross Reference Import tool (btsxrefimport.exe), which you can find in the BizTalks installation folder. The utility takes in a parameter that specifies the filename of an XML file that basically contains references to eight other XML files, which contain the data that must be imported into the xref_* tables. When data is in the tables, the cross-referencing functoids can be used to look up values in the tables. For detailed information about the syntax of these XML files and how to import the data into the database tables, refer to http://msdn.microsoft.com/en-us/library/aa578674(BTS.70).aspx. Table 3.13 lists the cross-referencing functoids and their usage.

TABLE 3.13 The Cross-Referencing Functoids

Name	Description
Format Message	Returns a formatted and localized string using argument substitution and, potentially, ID and value cross referencing
Get Application ID	Retrieves an identifier for an application object
Get Application Value	Retrieves an application value
Get Common ID	Retrieves an identifier for a common object
Get Common Value	Retrieves a common value
Remove Application ID	Removes the identifier for an application object
Set Common ID	Sets and returns an identifier for a common object

You can see an example of the use of the cross-referencing functoids in the "Advanced Maps" section.

Advanced Functoids

The advanced functoids perform advanced operations on their input. Some of them return an output that can be used as the input for another functoid or sent directly to the destination schema. The use of others assists the compiler in creating the necessary XSLT. Table 3.14 lists the advanced functoids, their usage, and parameters.

TABLE 3.14 The Advanced Functoids

Name	Description
Assert	This functoid takes in three parameters. The first must be a Boolean value. If the map is compiled in Debug mode, an exception is thrown with the text from the second parameter if the first parameter is false. The third parameter is used as output. If the map is compiled in Release mode, the third parameter is always returned. This is useful for debugging your map, and just as with assertions in normal C# code, it allows you to make sure some conditions are always true.
Index	This functoid takes in a minimum of 1 parameter and a maximum of 100 parameters. The first parameter must be a node from the source schema. and the rest represent indexes in each level from the node that is the first parameter and back to the root node. The functoid returns the value in the node given by the first parameter and indexed by the values from the rest of the parameters. An example is provided in the "Advanced Maps" section.
Iteration	This functoid takes in one parameter, which must be a node in the source schema. The functoid returns the iteration of the node, meaning that it returns 1 for the first occurrence of the input node, 2 for the second occurrence, and so on.
Looping	This functoid takes in a minimum of 2 parameters and a maximum of 100 parameters. The input parameters must be nodes in the source schema and cannot be other functoids. The functoid has no functional purpose and therefore no output as such. You must connect the looping functoid to a record in the destination schema, and the map then generates one output record for each of the input records that correspond to the input parameters. An example of the Looping functoid is provided in the section "Advanced Maps" section.
Mass Copy	This functoid takes in one parameter, which must be a node from the source schema. The functoid must be connected to a node in the destination schema, and it copies recursively all the nodes below the source node to the destination node.
Nil Value	This functoid takes in one optional parameter. If specified, the parameter must be a Boolean. If not specified, a value of true is assumed. The functoid is connected to a node in the destination structure, and if a value of true is specified in the first parameter, the output node is created with a value of nil.
Record Count	This functoid takes in one parameter, which is a node from the source schema. The functoid returns the number of times the node appears in the input regardless of scope.

TABLE 3.14 The Advanced Functoids

Name	Description
Scripting	This functoid is used to execute some script. The functoid is configurable to either call a specific method in a .NET class in an assembly or to configure a script that is either Inline C#, Inline JScript .NET, Inline Visual Basic .NET, Inline XSLT, or Inline XSLT Call Template. The XSLT scripting options can be used to control creation of destination structure, because the XSLT has access to the entire source structure and the XSLT is inserted into the generated XSLT. The other script types can be used to perform operations on input values and they must output one value. The Scripting functoid is discussed in more detail in the "Advanced Maps" section. If you use a .NET language in your inline script, you have access to the .NET namespaces found at http://msdn.microsoft.com/en-us/library/aa561456(BTS.70).aspx.
Table Extractor	This functoid takes in two parameters. The first must be a Table Looping functoid, and the second is a number that indicates the column from the table to get the value from. The output from this functoid can then be sent to another functoid as input or directly to the destination schema. An example of the Table Extractor functoid is provided in the "Advanced Maps" section.
Table Looping	This functoid takes in a minimum of 3 parameters and a maximum of 100 parameters. The functoid builds a table in memory useful for creating records in the output that have some structure that is not present in the input. The first parameter is a scoping link from a node in the source schema. The second parameter is the number of columns that should be in the table, which corresponds to the number of fields to create in the output. The third parameter and all following parameters are values that can be used to build the table. An example of the Table Looping functoid is provided in the "Advanced Maps" section.
Value Mapping	This functoid takes in two parameters. The first must be a Boolean value, and the second parameter can be any value. The functoid outputs the second parameter if the first parameter was true. This functoid differs from the Value Mapping (Flattening) functoid in that this functoid does not attempt to flatten the input into the output.
Value Mapping (Flattening)	This functoid takes in two parameters. The first must be a Boolean value, and the second parameter can be any value. The functoid outputs the second parameter if the first parameter was true. This functoid differs from the Value Mapping functoid in that this functoid attempts to flatten the input into the output. This is useful when you have multiple Value Mapping functoids with outputs going to the same record, because otherwise two instances of the destination record will be created.

Third-Party Functoids

You can download several third-party functoids and use them in your solutions. Use your favorite search engine to locate them.

Advanced Maps

This section describes some advanced options for mapping and provides some more details on how mappings can be done. Specifically some scenarios of advanced usage of the functoids in combination are described.

Mapping Optional Fields

If the source document of a map has an optional field in it that you are mapping to a field in the destination schema, the XSLT actually handles this for you and creates the destination field only if the source field exists. This way you won't get an empty field in the output if the source field isn't there.

This is achieved by wrapping the creation of the destination field in an `if` statement, as shown in Figure 3.16.

```
<xsl:if test="OrderDate">
- <OrderDate>
    <xsl:value-of select="OrderDate/text()" />
  </OrderDate>
</xsl:if>
```

FIGURE 3.16 Resulting XSLT from mapping optional fields.

Had the `OrderDate` element in the source not been optional, the `OrderDate` field in the destination would always be created; and if the element at runtime had not been present, the output would have an empty element. The generated XSLT would be the same as shown in Figure 3.16 but without the enclosing `if` statement.

Note that if you try to map an optional field to a required field, you get a warning at compile time or when you validate the map.

Looping Functoid

The looping functoid can be used to give the compiler hints as to how to create the XSLT when you are mapping across different hierarchies in the source document. For example, consider the mapping in Figure 3.17.

The idea of the map is to create one `AddressInformation` record in the output for the `ShippingInformation` record in the source and one for the `BillingInformation` record.

The output, however, is shown in Figure 3.18.

This is clearly not the output you want, because the address information is mixed in one `AddressInformation` record. To fix this, you use the Looping functoid, as shown in Figure 3.19.

The output is now as expected, as shown in Figure 3.20.

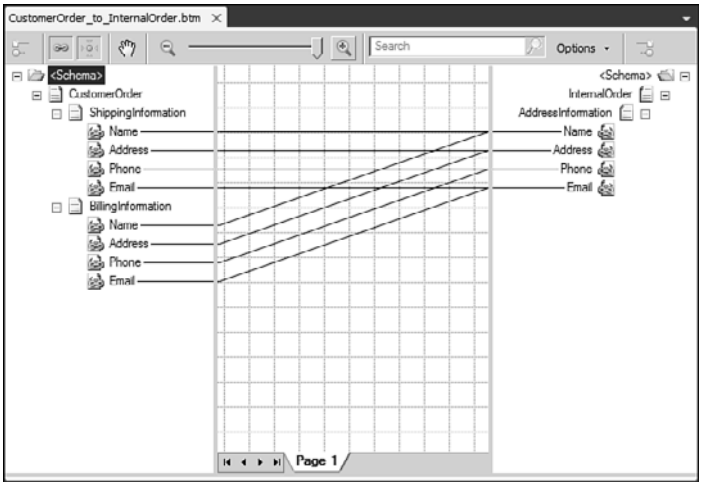

FIGURE 3.17 Mapping across hierarchies without Looping functoid.

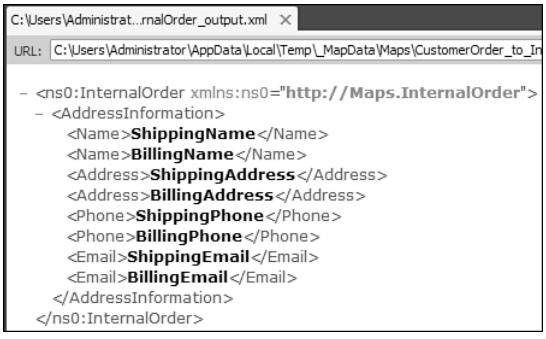

FIGURE 3.18 The output of the map when the Looping functoid is not used.

Note that in this example the `BillingInformation` record is mapped to an `AddressInformation` record before the `ShippingInformation`. This is because the order you add the inputs to the Looping functoid matters. If you add the `ShippingInformation` to the Looping functoid before the `BillingInformation`, the `ShippingInformation` is mapped first.

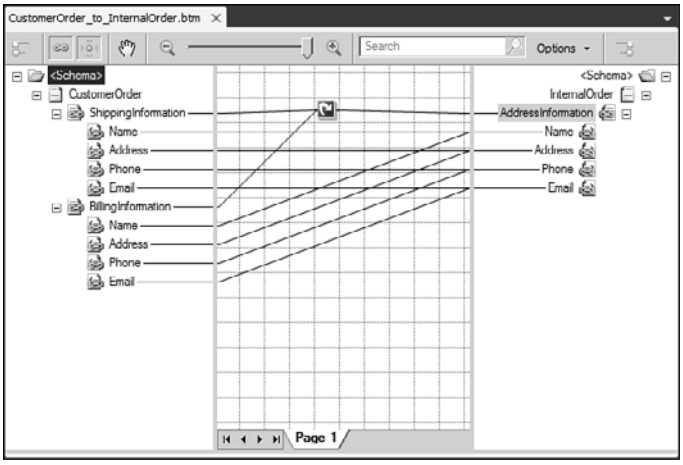

FIGURE 3.19 Mapping across hierarchies with Looping functoid.

FIGURE 3.20 The output of the map when the Looping functoid is used.

Index Functoid

The Index functoid provides a means to retrieve a specific value at a specific place in a hierarchy of fields.

If you have the XML shown in Figure 3.21 and you want to get the value `ItemNumber2` from the `ItemNumber` element in the second `OrderLine` in the first `Order` record, for example, you can use the Index functoid to do so.

This is done using the Index functoid as shown in Figure 3.22, with the parameters to the functoid being as shown in Figure 3.23.

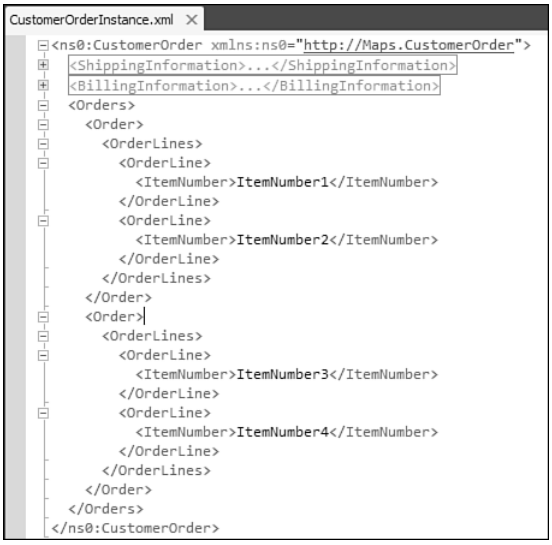

FIGURE 3.21 XML example of the Index functoid.

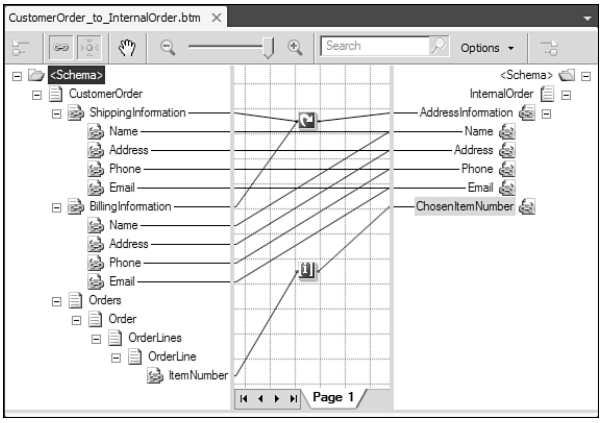

FIGURE 3.22 Using the Index functoid.

The parameters are first of all the field to get a value from and then the index of each parent level from that field and up. So, the parameters shown in Figure 3.23 are for the second `OrderLine` in the first `OrderLines` in the first `Order` record.

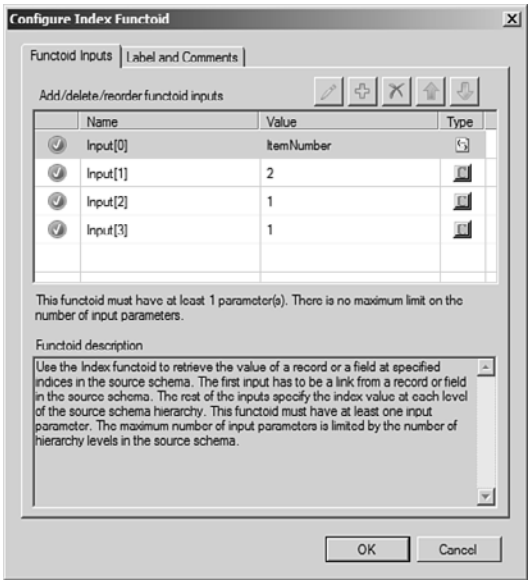

FIGURE 3.23 Parameters for the Index functoid.

Database Lookup

The Database Lookup functoids are used to look up values in a database if the data is not present in the source document of the map. This can be getting the price of an item or the address of a customer or anything else.

To explain the use of this functoid, consider the map depicted in Figure 3.24. You need to add the Database Lookup functoid first. This functoid takes four parameters:

- A lookup value. This is the value from the source document that you need to find information based on.
- A connection string to the database that holds the table you want to fetch information from.
- The name of the table you need to fetch information from. You can wrap the table name in square brackets ([and]) if you like (for instance, if spaces or reserved words are used in the table name).
- The column in the table that you want to match against the lookup value from the first parameter. This can also be wrapped in square brackets.

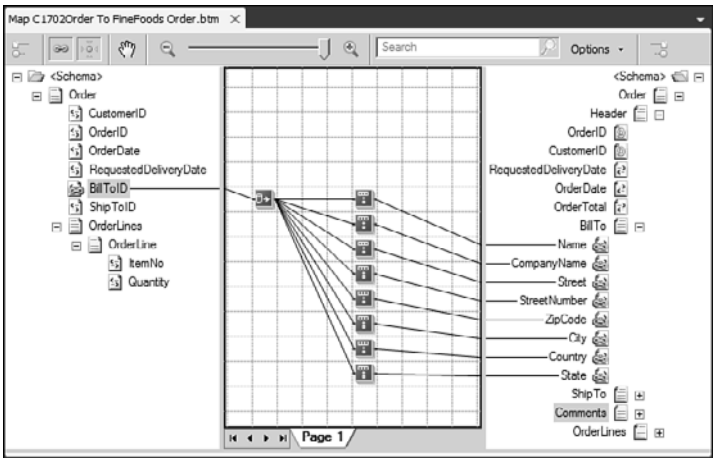

FIGURE 3.24 Using the Database Lookup functoid.

The functoid converts this into a SQL statement that looks like this:

```
SELECT * FROM <Param3> where <Param1> = <Param4>
```

In other words, it connects to the database using the connection string and then executes this SQL statement to get the row you need from the table you have specified.

TIP

If you are having trouble getting the connection string right, create an empty file on your desktop called conn.udl. Double-click it. A wizard will appear that helps you choose the provider, server, authorization scheme, and so on. When you have finished, save the work and open the .UDL file in a text editor to get the connection string.

The Database Lookup functoid returns only one row from the database. If the SQL statement returns more than one row, which is the case when the column specified in parameter four isn't a unique key in the table, the functoid just returns the first row from the data set returned by the SQL statement.

TIP

If you do not have one column that is unique, but need several, you can choose to use `[Col1]+[Col2]` as parameter four and then use a String Concatenate functoid to concatenate the fields from the source document that matches `Col1 + Col2`.

Because the Database Lookup functoid returns an entire row, you need to extract specific values from this row. In the example in Figure 3.24 there are eight Value Extractor functoids that will each extract a specific value from a column in the row.

A Value Extractor functoid takes in two parameters:

- The Database Lookup functoid that returns the row to extract a value from. Note that although the Database Lookup functoid is the only allowed input, the user interface allows you to use any Database functoid as the input for the Value Extractor functoid. This returns in runtime errors, however, so get this right at design time.
- The name of the column to extract. Note that this cannot be enclosed in square brackets (as you can with the table name and column name for the Database Lookup functoid, as discussed previously).

TIP

While developing a map that uses the Database Lookup functoid, you should add an Error Return functoid to your map. Let this functoid have the Database Lookup functoid as input, and let its output go to some element in the destination schema. This way, while developing and testing your map, you get any relevant error information that you can use to debug the connection. If you don't use the Error Return functoid, you will just have empty fields where the Value Extractor functoids should have put values.

Scripting Functoid

The Scripting functoid is used for two main things:

- To perform some task that the built-in functoids cannot do for you and which isn't needed often enough to justify developing a custom functoid. An example of this could be to generate a new *globally unique identifier* (GUID) or to do string replacement.
- To perform some task that the built-in functoids can do for you but that requires some combination of functoids that is too tedious to build. An example of this could be if-then-else functionality, which is described later in this section.

After dragging the Scripting functoid onto the map, you can double-click it to get to the functoid configuration window. Go to the Script Functoid Configuration pane, shown in Figure 3.25, where you can change the script.

In this screen, you may choose what language to use for your script. If you choose External Assembly, you can choose to call a public static method of a public class that is in a current assembly. The other five options allow for editing of the script to include in the map. For each, you can choose to import the script from an existing file by clicking the **Import from File** button. This is often a good idea because the editing window in the Script Functoid Configuration doesn't have IntelliSense, syntax highlighting, or even allow for the use of tabulators to indent code.

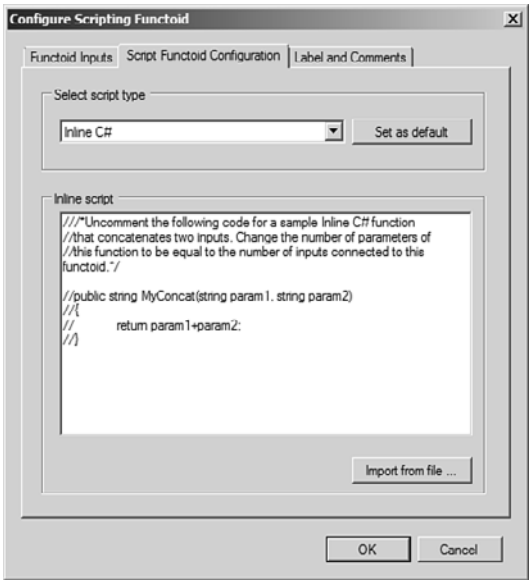

FIGURE 3.25 Scripting functoid.

The types of scripts that take parameters all take in strings. Therefore, you cannot send a node from the source document as a parameter and expect to treat it as an XmlNode. You only get the value of the node connected to the Scripting functoid.

Take care that the number of parameters your script takes matches the number of parameters you have provided to the functoid.

If you choose to leverage the power of one of the two XSLT scripting options and connect the Scripting functoid to a field in the destination schema, you take responsibility for creating the entire structure of the field you have connected the functoid to and any children of this field. The other four scripting types can output only a string, which is then copied to the field the Scripting functoid is connected to.

The XSLT scripting functoids are especially useful for performing tasks that deal with the source XML structure, because you have access to the entire source XML using XPath in the script, which you do not otherwise. Also, because you need to generate XML structures in the XSLT functoids, you have the possibility to create XML nodes for which there was no data support in the source XML. Assume, for instance, that you need to add an order line to all incoming orders that adds shipping expenses. You would need to copy all order lines from the incoming order to the destination document but also create a new order line to add to the existing ones. This is only doable in custom XSLT, be it either in a Scripting functoid or in a custom XSLT script that you use in your map instead of leveraging the Mapper.

Note that for XSLT you do not have access to all the nice features and functions of XSLT 2.0, because BizTalk only supports XSLT 1.0.

Functoid Combination

You can use the output from one functoid as the input for another functoid. This is useful for building functionality that doesn't exist in the built-in functoids.

If you want to make sure that a string in the input is trimmed for leading and trailing whitespace and also in uppercase, you need the functoids shown in Figure 3.26.

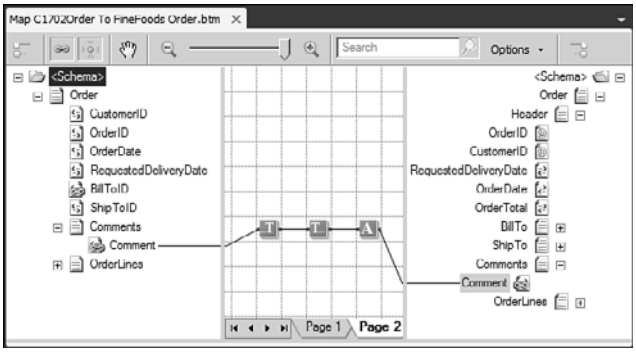

FIGURE 3.26 Functoid collection to trim input and convert it to uppercase.

Combination of Functoids for If-Then-Else

The built-in functoids provide you with two Value Mapping functoids that basically return their second parameter if the first parameter is true. This allows for an if-then solution, but there is no intuitive way of doing an if-then-else solution (that is, returning a third parameter if the first parameter is false). To build an if-then-else solution, you must use several functoids, as shown in Figure 3.27.

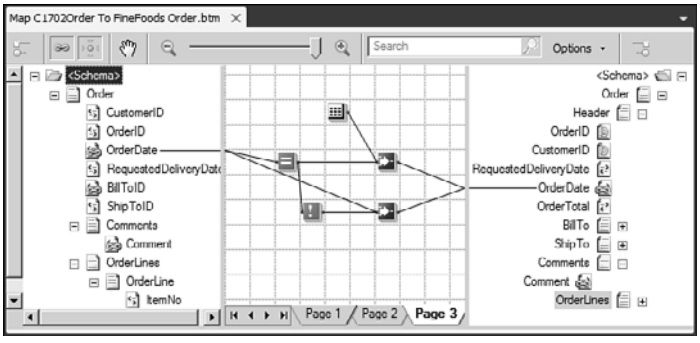

FIGURE 3.27 Performing the if-then-else logic.

The solution provided in Figure 3.27 is used to send the current date to the `OrderDate` field if the `OrderDate` provided in the input is empty. This is a case where the sender has a bug in his system that makes him send an empty element from time to time. The solution is to first use an Equal functoid to check whether the string equals the empty string

and use the output of this comparison as the first input for a Value Mapping functoid that takes in the current date as the second input. So if the field is empty, the current date is mapped. The output of the Equal functoid is also used as input to a Logical Not functoid, which negates the input. This is then used to allow another Value Mapping functoid to output the value of the `OrderDate` in case the Equal functoid did not return true, meaning the string wasn't empty.

Create Separated List

Assume that the order from a customer can have many Comment fields, but the internal order format allows for only one Comment field. In this case, you might want to concatenate the Comment fields from the input into one string for the output and separate the comments by some separator. Figure 3.28 shows an example of how to do this.

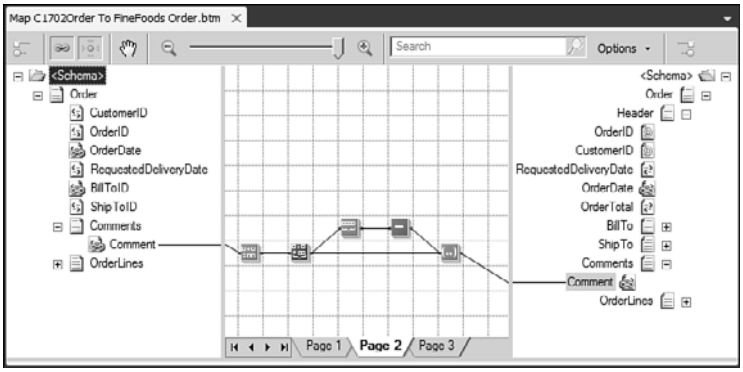

FIGURE 3.28 How to create a separated list of strings.

The functionality is built up using string functoids and one cumulative functoid. First, the input is concatenated with the separator. The output of this is sent to the Cumulative Concatenate functoid, which will then have the complete list as its output, with a separator at the end. This final separator is removed by using the String Extract functoid, which takes in the concatenated string as its first input. The second is the constant 1, and the third parameter is the length of the concatenated string minus 1.

Table Looping Functoid

The table looping functoid is useful to combine constants and fields from the source document into structures in the destination document. Let's revisit the challenge faced that was solved using the Looping functoid, as shown in Figure 3.19. Often, schemas that share a record for different addresses have a qualifier on the record, which contains information about what type of address the current record contains. So, the destination schema would probably be as shown in Figure 3.29.

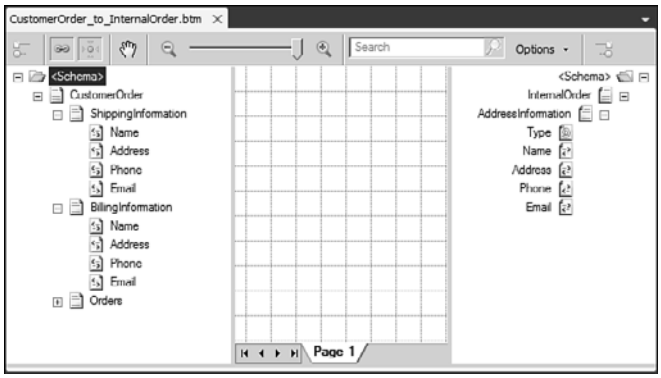

FIGURE 3.29 The map to implement before adding the Table Looping functoid.

The Table Looping functoid is used to build a table of information and then output each row in the table on at the time, thereby creating records in the destination schema. To implement the mapping in this example, first drag the functoid to the grid, and then drag a scoping record from the source to the functoid. In this case, that is the root node because that is the node that encompasses all the needed fields. The second parameter to the functoid must be a number indicating how many columns should be in the table. The third parameter and all the next parameters are values that can be used to build the table. As shown in Figure 3.30, all eight fields with information have been added as inputs to the functoid. Also, two constant strings have been added as parameters to the functoid (namely, the strings `Bill` and `Ship`). These two strings are the qualifiers used in this example, meaning that the `AddressInformation` record in the output that contains the shipping information must have a `Type` attribute with the value `Ship` and the other must have a `Type` attribute with the value `Bill`.

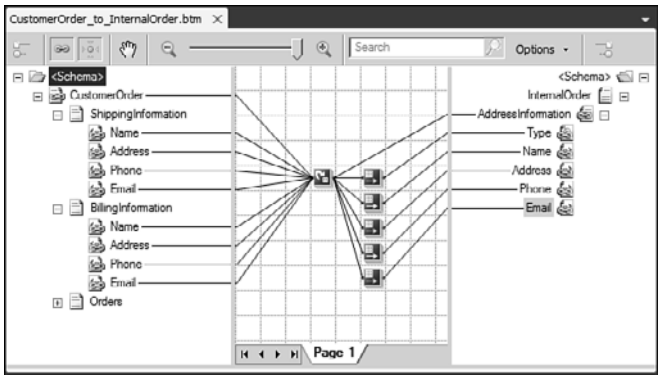

FIGURE 3.30 Using the Table Looping functoid.

To tell the compiler which record is to be created for each row in the table, you must drag the output of the Table Looping functoid to the record.

After doing this, you can start building the table, which is done either by double-clicking the functoid and then switching to the Table Looping Grid pane or by right-clicking the functoid and choosing **Configure Table Looping Grid**. This opens the table grid with as many columns as you have indicated by the second parameter to the functoid. You can now use the drop-down in each field to select which of all the parameters to the functoid to use for which columns in which rows of the table. The resulting table might look like Figure 3.31.

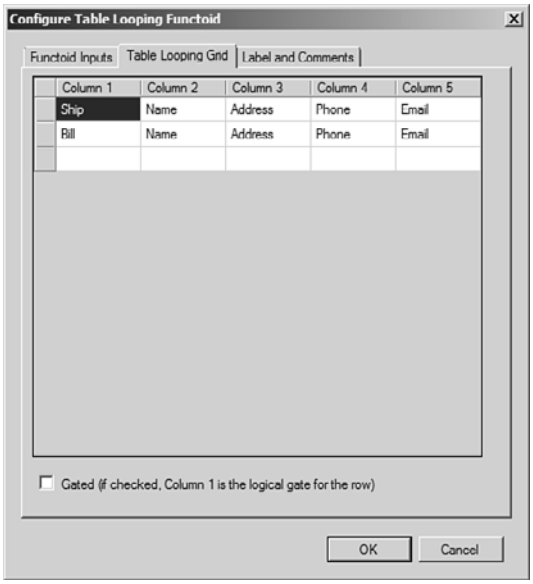

FIGURE 3.31 The table grid of the Table Looping functoid.

To determine which columns of the table go into which fields in the destination schema, you need to use the Table Extractor functoid. Add one of these, as shown in Figure 3.30, for each field to copy values into, and use the Table Looping functoid as the first input to each of them. The second parameter to each Table Extractor functoid must be the column number to extract. So, given the rows shown in Figure 3.31, let the first Table Extractor functoid have a second parameter of `1` and let its output go to the `Type` attribute of the `AddressInformation` record. Now configure the remaining four Table Extractor functoids to extract the correct column and map it to the correct field in the destination schema.

The resulting XML from testing the map should look as shown in Figure 3.32.

The grid configuration screen shown in Figure 3.31 has a check box at the bottom that can be checked to instruct the map that the data in the first column should act as a condition that specifies whether each row is created in the output. At runtime, the value of the first column is evaluated, and if data is found, the Value Extractor functoids associated with that row are called and the output record is created. If no data is found, the Value Extractor functoids are not called, and the record is therefore not created in the output. If

the input for the first column is a Logical functoid rather than a field, the output record is created if the value from the Logical functoid is true (and not otherwise).

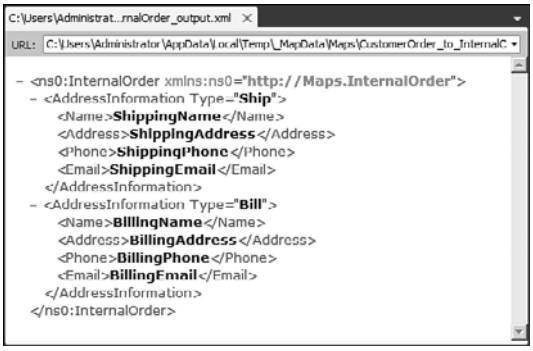

FIGURE 3.32 Result of using Table Looping functoid.

Conditional Creation of Output Nodes

You might sometimes want to implement conditional creation of output records. For instance, consider the opposite mapping of the one found in Figure 3.30. In this case, you will want to create a `ShippingInformation` record in case the `Type` attribute of the `AddressInformation` record has a value of `Ship` and similar with the billing information. This is achieved using logical functoids.

Logical functoids have a side effect to just being able to do logical operations. If you connect a logical functoid to an output record, the output record is only created if the functoid returns true. This means that the reverse mapping of the address information can be solved, as shown in Figure 3.33.

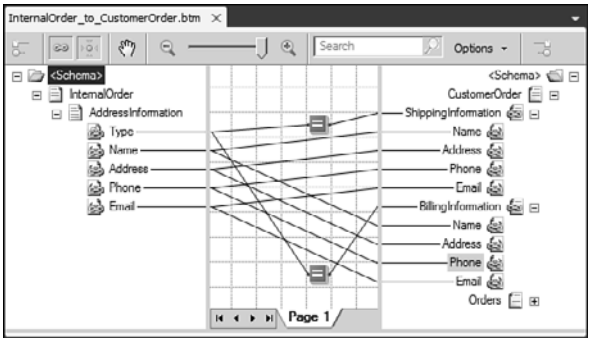

FIGURE 3.33 Conditional creation of records.

In this sample, the upper Equal functoid has a second parameter with a value of `Ship`, and the `ShippingInformation` record is therefore only created when the `Type` attribute has this value. The same applies for the `BillingInformation`, which is only created when the `Type` attribute has a value of `Bill`.

Custom XSLT

There are mappings that the Mapper cannot create for you using the built-in functoids. In that case, you have two options:

- Extend the map with Scripting functoids that perform the functionality you cannot accomplish with the built-in functoids.
- Create the map in the Mapper but don't use links or functoids. Instead, use a custom XSLT script for the functionality of the map.

Also, the XSLT generated by the Mapper is as good as it gets when things are automatically generated. If you know what you are doing, you can usually create yourself more efficient XSLT that performs better. If you have performance issues with your map, you might therefore also choose to write a custom XSLT script and bypass the Mapper.

If you choose the option of creating an entire XSLT script to use as the map, you need to use the Custom XSL Path and the Custom Extension XML properties of the map, as explained earlier. After creating your custom XSL and possibly a custom extension XML, you need to specify the path to the XSL and extension XML in the two properties. This effectively bypasses anything you might have done inside the Mapper.

TIP

As a starting point for your custom XSLT, you can get an XSLT with all the correct namespaces and definitions if you follow these steps: Add the map to your project and choose source and destination schemas. Then right-click the map in Solution Explorer and choose **Validate Map**. In the output window, you now get a link to the generated XSLT. Copy this file to your project and point your map to this file using the Custom XSL Path property. Edit the XSLT as needed.

The extension XML is some XML that is used to create a link between a namespace prefix that you can use in your custom XSLT and an external assembly that contains methods you want to call. Just as with the XSLT, you can get an example of a custom extension XML file when validating your map. You get an extension XML during this process only if you are actually calling an external assembly from the map you are validating.

Cross Referencing

Often you need to translate some numbering scheme into another during the execution of a map. An example of this is when mapping between customer order formats and your own format or between your own format and the format of your supplier. In this case, you might have an item number in your internal ERP system, and the customers and suppliers have their own numbers. The cross-referencing functoids help you achieve this. The setup is fairly simple: You create some XML files that contain the information about trading partners and the numbers that need to be translated and then you use an import tool to

import this XML into the cross-referencing-specific tables (xref_*) in the BizTalkMgmtDb database. The functoids can then be used to extract values from the tables.

This section contains a simple example, covering the scenario of item numbers that need to be translated from the numbers the customer uses to the numbers FineFoods uses. This scenario leverages only half of the XML documents that can be used for other cross-referencing features. For a complete description of the other XML files, refer to http://msdn.microsoft.com/en-us/library/aa578674(BTS.70).aspx.

This scenario leverages five XML documents. The first one is just a container that contains links to the other four. The XML files can be seen in Listing 3.2, Listing 3.3, Listing 3.4, Listing 3.5, and Listing 3.6.

LISTING 3.2 Contents of Cross-Referencing Setup File

```
<?xml version="1.0" encoding="UTF-8"?>
<Setup-Files>
    <App_Type_file>C:\CrossReferencing\ListOfAppType.xml</App_Type_file>

<App_Instance_file>C:\CrossReferencing\ListOfAppInstance.xml</App_Instance_file>
    <IDXRef_file>C:\CrossReferencing\ListOfIDXRef.xml</IDXRef_file>
    <IDXRef_Data_file>C:\CrossReferencing\ListOfIDXRefData.xml</IDXRef_Data_file>
</Setup-Files>
```

Listing 3.2 shows the contents of the setup file, which is really just a collection of links to the XML documents that contain the actual data that should be imported. Other than the four shown references, four other XML documents can be specified:

- ValueXRef_file
- ValueXRef_Data_file
- Msg_Def_file
- Msg_Text_file

These are not covered in this simple scenario.

LISTING 3.3 Contents of App_Type_file

```
<?xml version="1.0" encoding="UTF-8"?>
<listOfAppType>
  <appType>
    <name>ERP</name>
  </appType>
</listOfAppType>
```

Listing 3.3 shows the contents of the App_Type_file XML document. It is basically a list of application types that can exist. You can use any string you want. For this scenario only the application type `ERP` is used.

LISTING 3.4 Contents of App_Instances_file

```
<?xml version="1.0" encoding="UTF-8"?>
<listOfAppInstance>
  <appInstance>
    <instance>ERP_C1702</instance>
    <type>ERP</type>
  </appInstance>
  <appInstance>
    <instance>ERP_Internal</instance>
    <type>ERP</type>
  </appInstance>
</listOfAppInstance>
```

Listing 3.4 shows the contents of the App_Instances_file XML document. It is a list of instances of the application types from the App_Type_file XML document. The XML in Listing 3.4 has two instances of the `ERP` type, namely the ERP system from the customer and the internal ERP system.

LISTING 3.5 Contents of IDXRef_file

```
<?xml version="1.0" encoding="UTF-8"?>
<listOfIDXRef>
  <idXRef>
    <name>ItemID</name>
  </idXRef>
</listOfIDXRef>
```

Listing 3.5 shows the contents of the IDXRef_file XML document. It is a list of types of IDs that need to be translated. In this scenario, we need to translate identifications of items, but this can be any string you specify.

LISTING 3.6 Contents of IDXRef_Data_file

```
<?xml version="1.0" encoding="UTF-8"?>
<listOfIDXRefData>
  <idXRef name="ItemID">
    <appInstance name="ERP_C1702">
      <appID commonID="ITEM1">123</appID>
      <appID commonID="ITEM2">456</appID>
      <appID commonID="ITEM3">789</appID>
```

```
    </appInstance>
    <appInstance name="ERP_Internal">
      <appID commonID="ITEM1">4301</appID>
      <appID commonID="ITEM2">4398</appID>
      <appID commonID="ITEM3">5432</appID>
    </appInstance>
  </idXRef>
</listOfIDXRefData>
```

Listing 3.6 shows the contents of the IDXRef_Data_file XML document. It is the actual values that can be translated. In this scenario, the value `123` as an item identification from customer `C1702` is translated into `4301`, which is the corresponding item identification in the internal ERP system.

The functoids used to do the translation are shown in Figure 3.34.

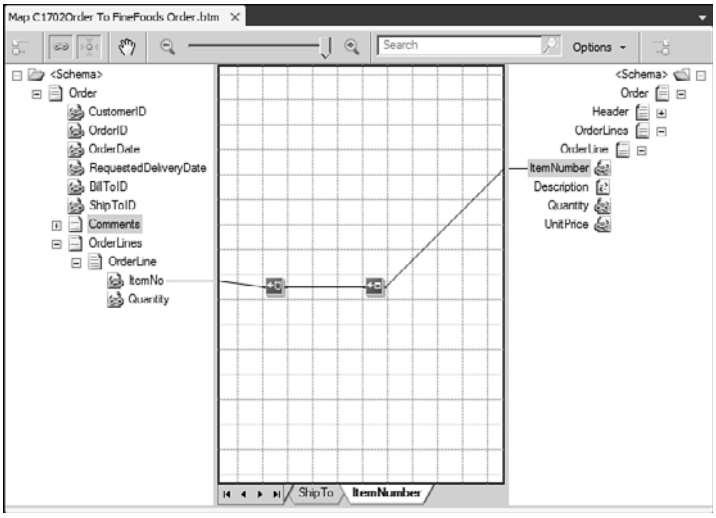

FIGURE 3.34 Using the cross-referencing functoids.

Figure 3.34 shows how to translate the item identification from the `ItemNo` field in the order from customer C1702 to the `ItemNumber` in the destination schema. The `ItemNo` field is mapped to a Get Common ID functoid, which has these three parameters:

- The constant string `ItemID`, which matches the type of ID to convert, as specified in the IDXRef_file XML document
- The constant string `ERP_C1702`, which matches the application instance of an ERP system, as found in the App_Instances_file XML document
- The `ItemNo` field

The functoid uses the value form the source to retrieve the common ID for this application-specific ID. For instance, a value in the source document of `123` returns a common ID of `ITEM1`. This value is then passed into a Get Application ID functoid, which also has three parameters. The first parameter is the same as for the Get Common ID functoid. The second is the constant string `ERP_Internal`, which tells the functoid to get the ID for this particular application instance. The third is the output of the Get Common ID functoid. For this scenario, the third parameter has a value of `ITEM1`, and the output of the functoid is the string `4301`.

Building Custom Functoids

You can use the built-in functoids as building blocks to provide for most mapping needs you will encounter. Sometimes, however, you will need functionality that the built-in functoids cannot provide. Other times, you might find yourself building the same combination of functoids to solve a specific problem in your map over and over again, which is tiresome and which creates a mess in your map. To overcome this, you can develop your own functoids. Developing custom functoids is not nearly as scary as it sounds, and in fact you will probably find out that the most difficult part is to create an icon that is nice and descriptive of your functoid. This section describes how to develop a custom functoid.

Functoids are divided into two categories:

- The noncumulative, or "normal," ones, which expose one method that takes in some parameters or possibly no parameters and returns a string.
- The cumulative ones, which expose three methods, where the first is called to initialize the functoid; the second is then called for all inputs, and the third method is called to get the final value.

Also, functoids can be divided into two other types:

- The type that is compiled into an assembly and put into the Global Assembly Cache (GAC). It exposes one or more methods that are called at runtime to perform your functionality. This type is also known as a referenced functoid.
- The type that doesn't expose a method that is called at runtime, but instead outputs a script that is included in the map. This type is also known as an inline functoid.

You should consider developing an inline functoid when

- You have no worries about your code getting into the map as clear text (which allows others to read it and possibly modify it).
- Your functoid depends only on .NET namespaces that are available to maps. For a full list of these, refer to http://msdn.microsoft.com/en-us/library/aa561456(BTS.70).aspx.
- You do not want to have to maintain another assembly, remembering to add it to installation scripts, deploying it to all servers in your BizTalk group, and so on.

- You want to provide the developer that uses the functoid with the ability to debug the maps that use your functoid.
- You are developing more than one functoid, and they need to share variables.

You should consider developing a referenced functoid when

- You want to be able to put a new assembly in the GAC and restart host instances for it to work in all maps that use the functoid without any need to maps to be recompiled.
- You do not want your business logic code exposed in clear text for all to read and possibly modify.
- Your functoid depends on .NET namespaces that are not available to maps.

You do not have to choose either an inline functoid or a referenced functoid. As a matter of fact, you can develop your functoid to be both and let the developer of the map choose which implementation to use.

Initial Setup

No matter what type of functoid you want to create, you want to create a Visual Studio 2010 project for it. You do this by either right-clicking your solution and choosing **Add**, **New Project** or by creating a new project in a new solution if you do not have an existing solution you want to add the project to. You can have the project with your functoids in the same solution as your BizTalk projects and any other projects, if you like. The project should be a Class Library project, as shown in Figure 3.35.

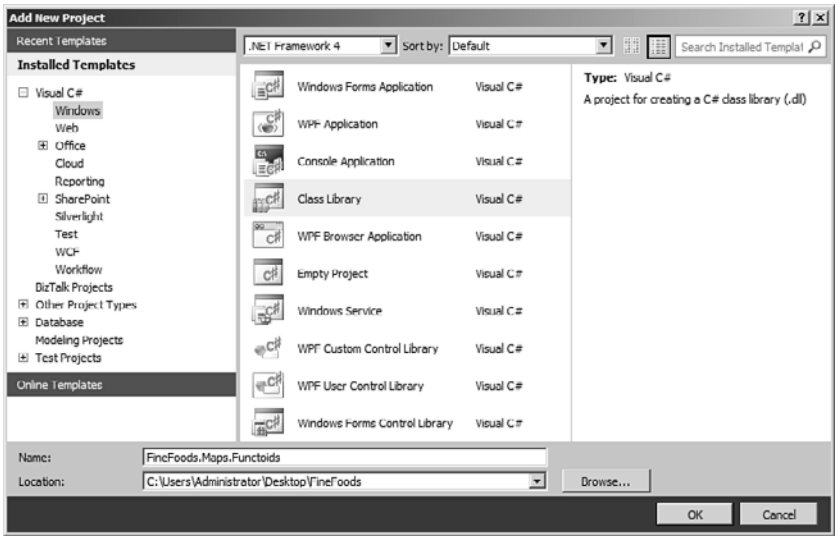

FIGURE 3.35 Adding a new Class Library project to your Visual Studio 2010 solution.

After adding the project to your solution, you should either rename the automatically added class to a better name or delete it and add a new one. Also, you need to add a reference to the Microsoft.BizTalk.BaseFunctoids.dll, which you can find under <InstallationFolder>\Developer Tools. Finally, you need to add a string name to the assembly so that it can be GAC'ed after the functoid has been developed.

The `Microsoft.BizTalk.BaseFunctoids` namespace contains a `BaseFunctoid` class that must be the base class for all functoids. You must therefore let your class inherit from this and call the constructor of the base class. Listing 3.7 shows an example.

LISTING 3.7 Extending the Needed Base Class Required to Create a Custom Functoid

```
using Microsoft.BizTalk.BaseFunctoids;


namespace FineFoods.Map.Functoids
{
    public class StringReplace : BaseFunctoid
    {
        public StringReplace()
            : base()
        {
        }
    }
}
```

Inside the constructor, you need to set the value of some properties and call some methods on the base class. The steps you must go through for all functoids are described in this section, and the ones that are specific for either normal or cumulative functoids are described in the next sections:

1. Add a resources file to your project. To do so, right-click your project and choose **Add**, **New Item**. In the next screen, choose **Resources File**. Provide a descriptive filename that reflects whether the resources file is for one functoid only or for a collection of functoids that are in the same assembly. Figure 3.36 shows an example.

NOTE

For functoids, adding a resources file is not optional like it is for developing pipeline components, where it is still best practice to do so. The base class needs a resource file for the values for the functoid name, tooltip, description, and icon.

2. Add three string resources: one for the name of the functoid, one for the tooltip of the functoid, and one for the description of the functoid. Provide descriptive names of the resources. Figure 3.37 shows an example.

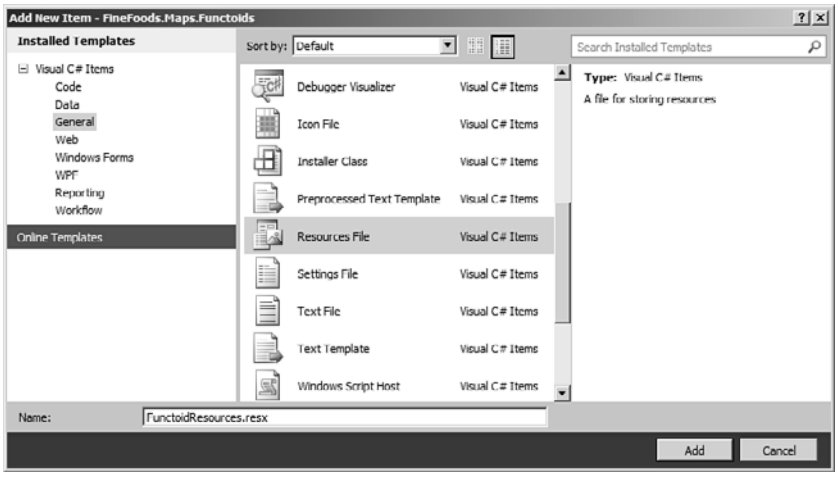

FIGURE 3.36 Adding a resources file to your project.

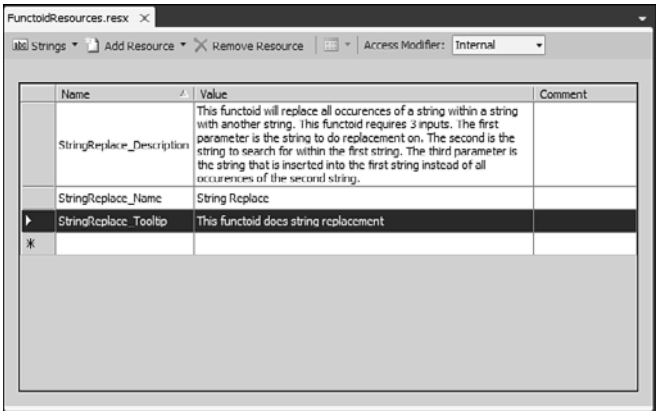

FIGURE 3.37 Adding resources for name, tooltip, and description..

3. Add an image resource of type `Bitmap` for the icon of the functoid. Provide a descriptive name of the resource. After adding it, you can edit the bitmap. Change it to be 16x16 pixels in the properties of the bitmap, and then release your inner artist. Figure 3.38 shows how to add the bitmap resource.
4. Assign a value to the `ID` property. The value must be an int, and it must be greater than 6000 because the first 6000 are reserved for internal BizTalk usage. Always keep track of all IDs you use in your organization to make sure you do not get an overlap. If you use third-party functoids, there is no way of knowing what other IDs are in use by these other than using reflector on the assemblies and looking at the source code.

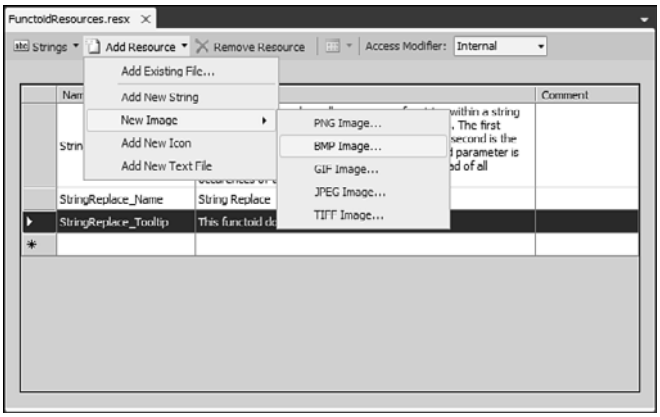

FIGURE 3.38 Adding a bitmap resource to serve as the icon for the functoid.

5. Call the `SetupResourceAssembly` method to let the base class know where to find the resources file that contains the resources for name, tooltip, description, and icon. The method takes two parameters. The first is the fully qualified .NET type name of the resources file. This is normally the name of the project concatenated with a period and the name of the resources file without extension. So if your project is called FineFoods.Maps.Functoids and your resources file is called FunctoidResources.resx, the fully qualified .NET type name of the resources file will be FineFoods.Map.Functoids.Resources. If in doubt, open <ResourceFile>.Designer.cs file, where <ResourceFile> is the name of your resources file without the .resx extension. In the designer file, you can see namespace and class name. Concatenate these with a period in between and you have the fully qualified name. The second is the executing assembly. Listing 3.8 shows an example.
6. Call the `SetName` method, which takes in one string parameter that defines the name of the resource that the base class should use to find the name of the functoid.
7. Call the `SetTooltip` method, which takes in one string parameter that defines the name of the resource that the base class should use to find the tooltip of the functoid.
8. Call the `SetDescription` method, which takes in one string parameter that defines the name of the resource that the base class should use to find the description of the functoid.
9. Call the `SetBitmap` method, which takes in one string parameter that defines the name of the resource that the base class should use to find the icon of the functoid.
10. Set the value of the Category property. This property is of the type `FunctoidCategory`, which is an enumeration, and it must be set to match the category this functoid should belong to. There are 25 different categories, of which only 7 are supposed to be used in custom functoids. These are Conversion, Cumulative, DateTime, Logical, Math, Scientific, and String.

    As you can probably imagine, these are used to let the Mapper Toolbox know in which group of functoids to show your custom functoid. Some of the categories are

also used to let the Mapper know how to create the XSLT; that is, functoids in the category Logical are useful for determining when to create destination nodes. This is explained in more detail later.

11. Determine how many parameters your functoid should take in as a minimum and as a maximum. Call the `SetMinParams` and `SetMaxParams` with the correct values.
12. For each parameter, determine what the source of the input link for the parameter can be. For custom functoids, the parameters can often be links coming from anything that has a value. After determining what possible inputs the parameters can have, you must call `AddInputConnectionType` for each parameter in the order of the inputs to specify what possible inputs the parameters can have. The possible parameter values for the `AddInputConnectionType` method call are the values in the `ConnectionType` enumeration, and there is a value for each functoid category and also some other possible values like `All`, `AllExceptRecord`, `Element`, and so on. The `AllExceptRecord` is often used because this will allow all inputs that are not a record, and this can be useful because a record does not have a value, whereas others have a value.
13. Determine what the outgoing link from the functoid can be connected to. After determining this, set the appropriate value to the `OutputConnectionType` property, which is of type `ConnectionType` enumeration.

NOTE

The tooltip you add to your functoid is actually not used anywhere. Microsoft is aware of this and will look into it for future releases. This has two implications. The first implication is that you should still add the tooltip and provide valid values for it so that your functoid will look fine in the next versions, as well. The second implication is that you don't need to worry when you don't see the tooltip of your custom functoid anywhere.

For setting either the input connection type or the output connection type, you can set it to a combination of values, which gives you control of what possibilities you want to allow.

Listing 3.8 shows a functoid that does a string replacement on an incoming string. The functoid code is not complete, but contains the methods and properties that have been discussed up to now.

LISTING 3.8 Functoid That Does String Replacement

```
public class StringReplace : BaseFunctoid
{
    public StringReplace() : base()
    {
        ID = 8936;

        SetupResourceAssembly(GetType().Namespace + ".FunctoidResources",
```

```
            Assembly.GetExecutingAssembly());

        SetName("StringReplace_Name");
        SetTooltip("StringReplace_ToolTip");
        SetDescription("StringReplace_Description");
        SetBitmap("StringReplace_Icon");

        Category = FunctoidCategory.String;
        SetMinParams(3);
        SetMaxParams(3);

        AddInputConnectionType(ConnectionType.AllExceptRecord);
        AddInputConnectionType(ConnectionType.AllExceptRecord);
        AddInputConnectionType(ConnectionType.AllExceptRecord);

        OutputConnectionType = ConnectionType.AllExceptRecord;
    }
}
```

Normal Functoid

The functoid code in Listing 3.8 is not finished yet. It still has no implementation of the functionality it is supposed to do. As explained earlier, this can be achieved either as a referenced functoid or as an inline functoid and either as a normal functoid or as a cumulative functoid. Implementing it as both a referenced functoid and as an inline functoid is explored in this section.

Referenced Functoid

When implementing a functoid as a referenced functoid, you must provide the method that is to be called at runtime. Listing 3.9 shows a method that provides the string replacement functionality.

LISTING 3.9 A Method Performing String Replacement

```
public string Replace(string str, string search, string replace)
{
    if (String.IsNullOrEmpty(str))
        return String.Empty;
    if (String.IsNullOrEmpty(search))
        return str;
    return str.Replace(search, replace);
}
```

To instruct BizTalk what method to call at runtime, you must call the `SetExternalFunctionName` method. This method has two overloads.

- The first takes in three parameters, where the first is the full name of the assembly, the second is the class that contains the method, and the third is the name of the method to call.
- The second takes in four parameters but is not intended to be used in custom code.

Listing 3.10 shows the added functionality to the functoid from Listing 3.8 that is needed to make the functoid work.

LISTING 3.10 A Referenced Functoid That Does String Replacement

```
public class StringReplace : BaseFunctoid
{
    public StringReplace()
        : base()
    {
        // Functionality from Listing 3.8
        SetExternalFunctionName(GetType().Assembly.FullName,
            GetType().FullName,
            "Replace");
    }

    public string Replace(string str, string search, string replace)
    {
        if (String.IsNullOrEmpty(str))
            return String.Empty;
        if (String.IsNullOrEmpty(search))
            return str;
        return str.Replace(search, replace);
    }
}
```

The call the `SetExternalFunctionName` in Listing 3.10 is coded to have the method to call inside the same class as the functoid itself. If this is not the case, you must change the parameters to point to the correct assembly, class, and method.

TIP

If your functoid needs to check whether an input parameter is either a valid numeric value or a valid date, the `BaseFunctoid` class provides static methods you can use for this so you do not need to implement this yourself. You can just use `BaseFunctoid.IsNumeric(stringparameter)` to check for numeric values and `BaseFunctoid.IsDate(stringparameter)` to check for date values. Both methods return a Boolean, and the `IsNumeric` method can optionally have a ref parameter that will contain the converted value.

Inline Functoid

If you should choose to implement an inline functoid rather than a referenced functoid, you should not call the `SetExternalFunctionName` method, but instead some other methods and properties must be used.

The first is a method called `AddScriptTypeSupport`. It takes in one parameter, which is the `ScriptType` enumeration. You must send in the value that matches the script type you will be creating. For instance, you can send in a value of `ScriptType.CSharp` to tell the Mapper that the script is a C# script.

The second is a method called `SetScriptBuffer`, which is called to set the script that will be included in the map. It takes in two parameters and one optional parameter. The first parameter is the `ScriptType` for this script. The second parameter is the string that is the actual script. Most often, this parameter is a method call to a method that returns the string that contains the script and not a constant string itself, because that would be too big and confusing. The third and optional parameter is used for cumulative functoids, which are described in the next section.

The third method used for inline functoids is called `SetScriptGlobalBuffer`. This method is used to add some script that must be global for all your scripts. This can initialize a variable, for instance, which is needed for cumulative functoids or functoids that just need to know values from other scripts. Just as the `SetScriptBuffer` method, this method takes in the `ScriptType` of the script and a string that contains the script.

The fourth is a property called `RequiredGlobalHelperFunctions`. This is used to let the Mapper know whether some built-in helper functions are needed for the script to execute. This is to allow for the use of the built-in `IsNumeric` and `IsDate` methods that are easily accessible in a referenced functoid. Also, for inline functoids, you can make use of the `ValToBool` method, which tests whether your string is a Boolean value. This method is not accessible for referenced functoids.

NOTE

When developing custom inline functoids, you can add inline script for as many of the supported languages as you want. Just call `AddScriptTypeSupport` for the appropriate script type and call `SetScriptBuffer` to set the appropriate script. Which one is chosen by the map is dependent on the Script Type Precedence property you can set as a property on the map grid.

Listing 3.11 shows a method that generates the same method as shown in Listing 3.9, only for an inline functoid.

LISTING 3.11 An Inline Functoid That Does String Replacement

```
private string GetCSharpBuffer()
{
    StringBuilder sb = new StringBuilder();
    sb.Append("\n");
    sb.Append("public string Replace(string str, string search, string
    ➥replace)\n");
    sb.Append("{\n");
    sb.Append("\tif (String.IsNullOrEmpty(str))\n");
    sb.Append("\t\treturn String.Empty;\n");
    sb.Append("\tif (String.IsNullOrEmpty(search))\n");
    sb.Append("\t\treturn str;\n");
    sb.Append("\treturn str.Replace(search, replace);\n");
    sb.Append("}");
    return sb.ToString();
}
```

So, as you can see, for inline functoids, you must create a string that contains the exact same C# method you would have written if it were to be called in a referenced functoid. The new lines and tabulator characters are not needed, but are there to make the method look readable when viewing it inside the map after you have compiled it.

For inline functoids, you have the option of generating an inline script that takes in a variable number of inputs. This requires some other method calls and is described in the "Advanced Functoids" section.

TIP

The easiest way to create a method to be used in an inline functoid is to create the method as a normal method first and test your functoid as a referenced functoid. Once the functionality is as you want it to be, you can do a string replacement on the method, replacing all quotation marks with escaped quotation marks. Then you just cut and paste the lines from the method one at the time to a new method where you append to the `StringBuilder` and change the functoid to be an inline functoid instead.

The code for the inline functoid that does string replacement can be seen in Listing 3.12.

LISTING 3.12 An Inline Functoid That Does String Replacement

```
public class StringReplace : BaseFunctoid
{
    public StringReplace()
        : base()
    {
       // Functionality from Listing 3.8
        AddScriptTypeSupport(ScriptType.CSharp);
        SetScriptBuffer(ScriptType.CSharp, GetCSharpBuffer());
    }

    private string GetCSharpBuffer()
    {
        StringBuilder sb = new StringBuilder();
       // Code to build the method, as shown in Listing 3.11.
        return sb.ToString();
    }
}
```

Creating an inline C# functoid is most of the times the most appropriate over XSLT for three reasons:

- You get the .NET framework, which enables you to write your functionality with a minimum number of lines.
- XSLT has lots of quotation marks, which can get heavy to track when building a string. For information purposes, the code that is needed in an XSLT Call-Template functoid for string replacement is shown in Listing 3.13. The reason for this quite long code in XSLT is that the version of XSLT that BizTalk supports does not include a native Replace function, so you have to do it yourself. Imagining the code to build this as a string for an inline functoid is left to the reader.
- The real strength of XSLT functoids is that XSLT can access the entire structure of the source schema and it has the responsibility of creating the output structure. This means that the functoid will be hard wired to those two structures. Because the purpose of a custom functoid is to take some functionality that is often needed and wrap it in a common generic component, custom XSLT functoids actually go against this purpose.

LISTING 3.13 XSLT Example of Doing String Replacement

```
<xsl:template name="MyXsltReplaceTemplate">
  <xsl:param name="str" />
  <xsl:param name="search" />
  <xsl:param name="replace" />
  <xsl:element name="Field6">
```

```
    <xsl:call-template name="DoReplace">
      <xsl:with-param name="str" select="$str" />
      <xsl:with-param name="search" select="$search" />
      <xsl:with-param name="replace" select="$replace" />
    </xsl:call-template>
  </xsl:element>
</xsl:template>

<xsl:template name="DoReplace">
  <xsl:param name="str" />
  <xsl:param name="search" />
  <xsl:param name="replace" />
  <xsl:choose>
    <xsl:when test="contains($str, $search)">
      <xsl:value-of select="substring-before($str, $search)" />
      <xsl:value-of select="$replace" />
      <xsl:call-template name="DoReplace">
       <xsl:with-param name="str" select="substring-after($str, $search)" />
       <xsl:with-param name="search" select="$search" />
       <xsl:with-param name="replace" select="$replace" />
      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="$str" />
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```


Cumulative Functoid

Cumulative functoids are useful for performing functionality on recurring elements. The built-in cumulative functoids provide functionality for finding the smallest number, greatest number, and so on of a reoccurring element.

If you want to develop a cumulative functoid yourself, you need to specify three methods rather than one. The first method initializes the functoid at runtime. The second is called for each occurrence of the recurring element, and the last is called at the end to retrieve the aggregated value that should be output in the end.

Thread Safety

For cumulative functoids, an issue of thread safety arises that is usually not present for normal functoids.

Normal functoids have just one method that is called, and unless you use some variable inside your method that is globally defined, you are usually safe. Cumulative functoids are different, though, because you need a global variable in your functoid to hold the state of

your calculations across the initialization, multiple calls to the add method, and the final call to the get method.

You get some help in making your functoid thread safe, though. To all three methods, an index variable is sent as a parameter, which is unique for each instance of the functoid inside the map, which means that you can use it as an identifier into some data structure you must maintain with the calculations you perform. Listing 3.14 shows an example of how the XSLT looks when a cumulative functoid is used multiple times in a map. You can see that the first occurrence of the cumulative functoid uses an index of `0` and the second occurrence uses an index of `1`.

LISTING 3.14 XSLT Generated When a Cumulative Functoid Is Used Two Times in One Map

```
<xsl:template match="/s0:InputRoot">
  <ns0:OutputRoot>
    <xsl:variable name="v1" select="userCSharp:Init(0)" />
    <xsl:for-each select="/s0:InputRoot/Field1">
      <xsl:variable name="v2" select="userCSharp:Add(0,string(./text()),"1000")" />
    </xsl:for-each>
    <xsl:variable name="v3" select="userCSharp:Get(0)" />
    <Feld1>
      <xsl:value-of select="$var:v3" />
    </Field1>
    <xsl:variable name="v4" select="userCSharp:Init(1)" />
    <xsl:for-each select="/s0:InputRoot/Field2">
      <xsl:variable name="v5" select="userCSharp:Add(1,string(./text()),"1000")" />
    </xsl:for-each>
    <xsl:variable name="v6" select="userCSharp:GetCumulativeMax(1)" />
    <Field2>
      <xsl:value-of select="$var:v6" />
    </Field2>
  </ns0:OutputRoot>
</xsl:template>
```

This way of using an index is the same both for a cumulative referenced functoid and a cumulative inline functoid. The scope parameter to the second method is not used and can therefore be ignored in your code.

Cumulative Referenced Functoids

For referenced functoids, the runtime engine doesn't necessarily instantiate an object of your functoid class for each map it executes, but rather reuses the existing object if present. Unfortunately, this means that your functoid can get an index of `0` as parameter to any one of the methods from multiple instances of the map at the same time without your code being able to distinguish them from each other. This, in turn, means that it is impossible to develop a custom referenced cumulative functoid that is thread-safe, and this should therefore be avoided.

If you want to develop a custom referenced cumulative functoid, you need to set the three methods that are to be used at runtime by the mapper. This is done via the `SetExternalFunctionName`, `SetExternalFunctionName2`, and `SetExternalFunctionName3` methods. They set the initialization method, the accumulation method, and the get method, respectively. Listing 3.15 shows an example of the code needed. The code is given in full except for the code already listed in Listing 3.8 because it will make it easier to understand the code in Listing 3.16, which shows how to build he same functionality for an inline functoid.

LISTING 3.15 Sample Code of a Custom Referenced Cumulative Functoid

```
private Dictionary<int, string> myCumulativeArray = new Dictionary<int,string>();


public CummulativeComma() : base()
{
       // All the functoid setup code seen in Listing 3.8
       SetExternalFunctionName(GetType().Assembly.FullName, GetType().FullName,
"InitializeValue");
       SetExternalFunctionName2("AddValue");
       SetExternalFunctionName3("RetrieveFinalValue");
}

public string InitializeValue(int index)
{
    myCumulativeArray[index] = "";
    return "";
}

public string AddValue(int index, string value, string scope)
{
    string str = myCumulativeArray[index].ToString();
    str += value + ",";
    myCumulativeArray[index] = str;
    return "";
}

public string RetrieveFinalValue(int index)
{
    string str = myCumulativeArray[index].ToString();
    if (str.Length > 0)
        return str.Substring(0, str.Length - 1);
    else
        return "";
}
```

Cumulative Inline Functoids

Contrary to referenced cumulative functoids, you can develop a thread-safe inline cumulative functoid. This is because whereas the Mapper reuses the same object for referenced functoids, there is no object to reuse for an inline functoid because all the code is inline in the XSLT. Therefore, the data structure is not shared among multiple instances of the map, effectively making the index parameter, which is unique across multiple instances of the functoid in one map, enough to guarantee thread safety. This requires, naturally, that you develop the functoid using the index parameter to access a specific entry in the data structure.

Building a custom inline cumulative functoid basically requires the same three methods as for a referenced cumulative functoid. As with the referenced version, you need to initialize the needed data structure.

For setting the needed three methods that are used at runtime, you must call the `SetScriptBuffer` method three times, with a parameter indicating whether you are setting the initialization, adding, or retrieval method. For initializing the data structure, you must call the `SetScriptGlobalBuffer` method. Listing 3.16 shows sample code for a custom inline cumulative functoid, with the code from Listing 3.8 omitted.

LISTING 3.16 Inline Version of the Referenced Functoid from Listing 3.15

```
public CummulativeComma() : base()
{
  // All the functoid setup code seen in Listing 3.8
  SetScriptGlobalBuffer(ScriptType.CSharp, GetGlobalScript());
  SetScriptBuffer(ScriptType.CSharp, GetInitScript(), 0);
  SetScriptBuffer(ScriptType.CSharp, GetAggScript(), 1);
  SetScriptBuffer(ScriptType.CSharp, GetFinalValueScript(), 2);
}

private string GetFinalValueScript()
{
    StringBuilder sb = new StringBuilder();
    sb.Append("\npublic string RetrieveFinalValue(int index)\n");
    sb.Append("{\n");
    sb.Append("\tstring str = myCumulativeArray[index].ToString();");
    sb.Append("\tif (str.Length > 0)\n");
    sb.Append("\t\treturn str.Substring(0, str.Length - 1);\n");
    sb.Append("\telse\n");
    sb.Append("\t\treturn \"\";\n");
    sb.Append("}\n");
    return sb.ToString();
}
```

```
private string GetAggScript()
{
    StringBuilder sb = new StringBuilder();
    sb.Append("\npublic string AddValue(int index, string value, string
scope)\n");
    sb.Append("{\n");
    sb.Append("\tstring str = myCumulativeArray[index].ToString();");
    sb.Append("\tstr += value + \",\";\n");
    sb.Append("\tmyCumulativeArray[index] = str;\n");
    sb.Append("\treturn \"\";\n");
    sb.Append("}\n");
    return sb.ToString();
}

private string GetInitScript()
{
    StringBuilder sb = new StringBuilder();
    sb.Append("\npublic string InitializeValue(int index)\n");
    sb.Append("{\n");
    sb.Append("\tmyCumulativeArray[index] = \"\";\n");
    sb.Append("\treturn \"\";\n");
    sb.Append("}\n");
    return sb.ToString();
}

private string GetGlobalScript()
{
    return "private Dictionary<int, string> myCumulativeArray = new
Dictionary<int,string>();";
}
```


Developing Advanced Functoids

This section covers some advanced topics related to developing custom functoids.

Functoids with a Variable Number of Inputs

Sometimes you need to develop a functoid that should take in a variable number of parameters. For instance, the Addition functoid in the Math category takes in a variable number of parameters. Doing this is only supported for creating inline functoids and can therefore not be done with a custom referenced functoid.

To develop a custom inline functoid that takes a variable number of parameters, you must do this:

1. Set the property `HasVariableInputs` to `true`.
2. In the constructor, call `AddScriptTypeSupport` for each script type you support.
3. Override the `GetInlineScriptBuffer` method. Listing 3.17 shows an example. This method takes in three parameters:
   - A script type determining the type of script to return.
   - An integer determining the number of parameters your functoid will be getting.
   - A function number for use with cumulative functoids. Values can be 0, 1 and 2, for initializing, accumulating, and retrieving functions, respectively.
4. Set the `RequiredGlobalHelperFunctions` to reflect any global helper methods you may need, such as the `IsDate`, `IsNumeric`, and so on.
5. Use `SetScriptGlobalBuffer` to declare any global variables you may need. For cumulative functoids, you need to initialize some data structure that is used across the calls to the three functions.

LISTING 3.17 Generating a Functoid That Takes in a Variable Number of Parameters

```
protected override string GetInlineScriptBuffer(ScriptType sT, int numPar, int
func)
{
    if(ScriptType.CSharp == scriptType)
    {
        StringBuilder builder = new StringBuilder();

        builder.Append("public string MyFunction(");

        for(int i=0; i<numParams; i++)
        {
            if(i > 0)
                builder.Append(", ");

            builder.Append("string param" + i.ToString());
        }
        builder.Append(")\n");
        // Method body; Do what you need with the parameters.
        builder.Append("{\n");
        builder.Append("}\n");

        return builder.ToString();
    }
    return string.Empty;
}
```

The code in Listing 3.17 assumes this is not a cumulative functoid and therefore ignores the func parameter. Had this been for a cumulative functoid, the method would have to return one of three functions, given the value of the `func` parameter. Also, the method shown in Listing 3.17 works only for C#. If the developer of the map requires something else, you must extend the method to also support that `ScriptType` and return valid methods for that.

Functoid Categories

When assigning a functoid category to your functoid, you get to choose between 25 different categories, of which only 7 are supposed to be used in custom functoids. These are Conversion, Cumulative, DateTime, Logical, Math, Scientific, and String.

Assigning one of these categories to your functoid has some effects:

- The category maps to one of the categories in the Mapper Toolbox in Visual Studio 2010, so choose a category that matches where you want the functoid to be placed.
- Some functoids have restrictions as to what types of input they can have. For instance, a Value Extractor must have a Database Lookup functoid as its first input. The Database Lookup is actually a functoid category in itself; it just belongs to the Database group in the Toolbox. This means that how your functoid will be used in a map may therefore also influence what category you want to choose for it.
- Some categories imply some semantics other than just the two preceding bullets. For instance, a Logical functoid, as explained earlier, when connected to an output record determines whether the record should be created. You can therefore not create a functoid that is in the Logical category and use it to map a value of true or false to a field.

You can see all the possible values for the functoid category when developing custom functoids on the MSDN site at http://msdn.microsoft.com/en-us/library/microsoft.biztalk.basefunctoids.functoidcategory(BTS.70).aspx. Most of them are for internal use only, because they impose some semantics that you cannot control in your code.

Deployment of Custom Functoids

After developing a functoid, it must be deployed to be used. Deployment of a functoid can be divided into deployment on a development machine where a developer can then use the functoid in any maps created and deployment on a server that needs to be able to run the functoid at runtime.

For a server that needs to execute the functoid at runtime, the assembly containing the functoid must be put into the *Global Assembly Cache* (GAC) if the functoid is a referenced functoid. If the functoid is an inline functoid, no deployment is necessary.

For easy deployment, you can add the assembly with the functoid to your BizTalk application, which will deploy it along with the rest of the assemblies when the exported MSI package is installed on the server. To do this, follow these steps:

1. Open the BizTalk Server 2010 Administration Console.
2. Right-click your application and choose **Add**, **Resources**.
3. Click **Add**.
4. Browse your way to the assembly that contains the pipeline component and double-click it (or click it once and click **Open**).
5. In the File Type drop-down, choose **System.BizTalk:Assembly**.
6. Make sure the **Add to the Global Assembly Cache on MSI File Install (gacutil)** check box is checked.
7. Click **OK**.

The final screen should look like Figure 3.39

For deployment to a developer machine, the assembly must be copied into the <InstallationFolder>\Developer Tools\Mapper Extensions folder. After copying the assembly to this folder, you can add it to the Toolbox in Visual Studio 2010.

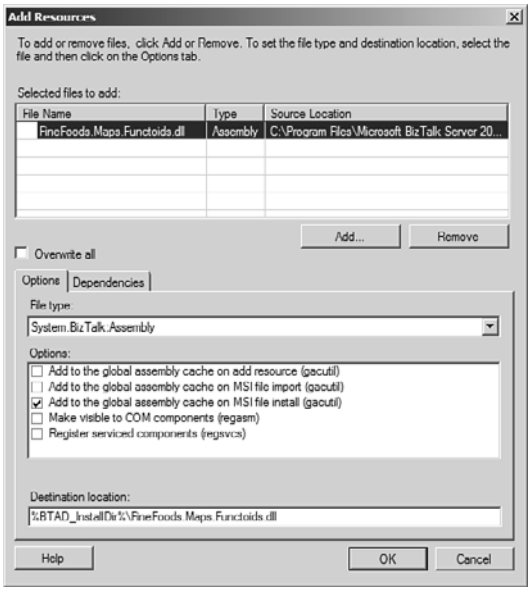

FIGURE 3.39 Adding an assembly to your BizTalk application to have it automatically deployed with the MSI file.

To deploy the assembly on a development machine for it to be used when developing maps, you should copy the assembly to the <InstallationFolder>\Developer Tools\Mapper Extensions folder. This is not strictly required, but it is the easiest way, because your functoid will then always be in the Toolbox, no matter how many times the Toolbox is reset. To add the component to the functoid Toolbox that is available at design time, follow these steps:

1. Open Visual Studio 2010
2. Go to **Tools**, **Choose Toolbox Items**, or right-click the Toolbox and choose **Choose Items**.
3. Go to the Functoids pane
4. If the functoid is present in the list, make sure it is checked.
5. If the functoid is not present in the list, click **Browse**, and browse your way to the assembly that contains the pipeline component.

If the functoid is placed in the <InstallationFolder>\Developer Tools\Mapper Extensions folder you can also right-click inside the Toolbox and choose **Reset Toolbox**. Be aware, though, that this completely resets the Toolbox, which might not be what you want.

Note that the Toolbox sometimes fails to update itself when new versions of a functoid are deployed. In this case, you can delete the following four files, which contain Visual Studio 2010s Toolbox setup:

- toolbox.tbd
- toolbox_reset.tbd
- toolboxIndex.tbd
- toolboxIndex_reset.tbd

On Windows Server 2008, using Visual Studio 2010, you can find these under C:\Users\<User>\AppData\Local\Microsoft\VisualStudio\10.0. For different operating systems or different installation options, you can search for them. They are hidden files, so you need to search for those. If you want to script the deletion of the files in a batch file, you can use the script shown in Listing 3.18 in a batch file.

LISTING 3.18 Script for Deleting Visual Studio 2010 Toolbox Items

```
@echo off
C:
cd "\Users\<User>\AppData\Local\Microsoft\Visual Studio\10.0"
del toolbox* /AH

pause
```

If you need to update a functoid that is also in the GAC, you must both update it in the GAC and in the <InstallationFolder>\Developer Tools\Mapper Extensions folder. This is

because Visual Studio 2010 uses the version in the GAC if present and any updates in the file system are therefore not used.

Note that overriding the assembly that has been copied to the <InstallationFolder>\Developer Tools\Mapper Extensions folder is not always possible because it might be in use by Visual Studio 2010, BizTalk runtime, or some isolated host like *Internet Information Services* (IIS). In these cases, you need to restart whatever programs are locking the assembly before you can copy the new version.

If you are trying to add your functoid to the Toolbox and get an error doing this, the most common causes are as follows:

- Your resources are not set up correctly, so the name, tooltip, description, and icon cannot be correctly fetched and used by the Toolbox.
- Your class is not marked public.
- There is an older version of your component in the GAC that is not valid.

If you get a sharing violation when copying your newly compiled assembly to the <InstallationFolder>\Developer Tools\Mapper Extensions folder, you might consider writing a script that deploys the new component for you. The script should

- Restart IIS using `iisreset`. Only needed in your script if you have any receive functionality running in IIS like hosted WCF services, HTTP, SOAP, or others.
- Restart BizTalks Host instance. Only needed if the functoid has been used by BizTalks runtime. This can be done in a couple of ways:
  - By using `net stop BTSSvc$BizTalkServerApplication` and `net start BTSSvc$BizTalkServerApplication`.
  - By using PowerShell and doing `get-service BTS* ¦ foreach-object -process {restart-service $_.Name}`
  - By using WMI. For details about this, refer to http://msdn.microsoft.com/en-us/library/aa578621(BTS.10).aspx.
- Copy the DLL to the <InstallationFolder>\Developer Tools\Mapper Extensions folder.
- Use `gacutil` to add the new version to the GAC, like this: `"C:\Program Files\Microsoft SDKs\Windows\v6.0A\Bin\gacutil.exe" /if NewlyCompiled.DLL`
- Delete the Toolbox items, as described earlier, if you are having issues updating the toolbox.

Visual Studio 2010 should probably be closed first, as well. You can do this a bit faster if you test your functoid in a Visual Studio 2010 BizTalk project that is the only project in a test solution. This Visual Studio 2010 will be the only instance of Visual Studio 2010 that has a lock on the file, and it can be closed and reopened faster than your entire solution.

Debugging

When developing a custom functoid, at some point you will probably want to debug it to make sure not only that it provides you with the expected output under normal circumstances but also to test borderline cases where input might be missing or be in an unexpected format. For a functoid that is to be used throughout your company, it is essential that all other developers can trust on your functoid to behave as expected and give the expected output under all circumstances.

How to debug a custom functoid depends on whether you have developed a referenced or an inline functoid.

Common Debugging Options for Inline and Referenced Functoids

The first and obvious way of debugging your custom functoid is to use it in a map and then validate and test your map from within Visual Studio 2010, checking that no errors occur and manually inspecting the output to make sure it provides the expected result.

Whether you need to debug a custom referenced or a custom inline functoid, you can and should leverage the unit testing capabilities of BizTalk maps. This basically enables you to check all cases the functoid should be able to handle. Because unit testing of maps should be enabled for all maps even if no custom functoids are present, it is an obvious testing solution for your custom functoids, as well. Unit testing of maps are explained in the "Testing" section.

Debugging Referenced Functoids

Debugging a custom referenced functoid can be achieved in different ways. No matter which way you choose, the functoid must be compiled in Debug mode.

Separate Assembly If the functionality the functoid is to perform is complex, you can write a separate assembly with the functionality and then a small console application that uses your assembly or do some standard unit tests on this to make sure the functionality is as expected. When satisfied with the results, you can wrap the functionality in a functoid either by copying the code or by referencing the unit-tested assembly.

Runtime Debugging Another way of debugging your functoid is to deploy a map that uses the functoid and debug it when an instance is sent through BizTalk. Sometimes your functoid actually depends on the context it is executed in, and in this case, this is your only option of debugging it.

When debugging the functoid inside BizTalks runtime, make sure the latest compiled version is in the GAC and make sure your host instance has been restarted, so it hasn't cached a previous version. Inside Visual Studio 2010, set breakpoints in your code wherever you want the code to break and allow you to inspect values and step into code. When this it done, a few steps are needed to start debugging:

1. Inside Visual Studio 2010, go to **Debug**, **Attach to Process**.

2. Choose the BTSNTSvc.exe process.
3. If the BTSNTSvc.exe process is not available, check
   - That the host instance is started
   - That you have checked **Show Processes from All Users** and **Show Processes in All Sessions**

   If multiple instances of the BTSNTSvc.exe process are running, you can stop the host instance that will be executing the map and notice what PIDs are active and then start the host instance. The new PID will be the one to attach to. Another option is to attach to all the BTSNTSvc.exe processes.
4. Click **Attach**.
5. Send a message through BizTalk

This causes BizTalk to load the functoid at runtime, and your debugging session breaks the runtime when one of your breakpoints is hit, allowing you to step through the code to debug your component.

DebugView A third way of debugging your custom referenced functoid is to leverage the `System.Diagnostics` namespace, which contains classes called `Debug` and `Trace`. In your code, you can insert statements that leverage these classes to write out either trace or debug statements, which are then viewable by tools like DebugView, which you can download from Microsoft's home page.

Listing 3.19 shows statements for leveraging the `Debug` and `Trace`.

LISTING 3.19 Leveraging the **`Debug`** and **`Trace`** Classes for Debugging

```
public string Replace(string str, string search, string replace)
{
    Trace.WriteLine("Replace method of \"String Replace\" functoid was
called.");
    Debug.WriteLine("Parameter str: " + str);
    Debug.WriteLine("Parameter search: " + search);
    Debug.WriteLine("Parameter replace: " + replace);

    if (String.IsNullOrEmpty(str))
    {
        Debug.WriteLine("First input was null or empty. Returning empty
string.");
        return String.Empty;
    }
    if (search == null)
    {
        Debug.WriteLine("Second parameter was null. Returning first
```

```
parameter.");
        return str;
    }
    Trace.WriteLine("Replace method of \"String Replace\" functoid has
ended.");
    str = str.Replace(search, replace);
    Trace.WriteLine("Replace method will return " + str);
    return str;
}
```


NOTE

When testing your map from within Visual Studio 2010, DebugView correctly shows any debug and trace statements you have in your code. When your map is executed at runtime by BizTalk, however, they are not. To rectify this, you must enable the **Capture**, **Capture Global Win32** option.

Debugging Inline Functoids

For inline functoids, you do not have the option of attaching to a process and setting breakpoints allowing you to step through your code because the code is inline in the map and doesn't use the assembly you compile.

Also, the option of using the `System.Diagnostics` namespace and leveraging the `Debug` and `Trace` classes will not work because the `System.Diagnostics` namespace is not one of the namespaces you can access from inline scripts.

When developing an inline functoid, it is impossible to know whether the script that is encoded in the string you output is actually a valid method that can compile. Therefore, it is often easiest to either develop your functoid as a referenced functoid or to develop the functionality needed in a separate assembly. Either way, you can debug that as mentioned earlier in the section about debugging custom referenced functoids, and once the functionality is as you want it to be, you can create a method that wraps the entire method in a string.

Testing of Maps

A map performs a transformation from one XML format into another. It is essential that the output generated is valid given the schema that describes the output, because otherwise you are sending invalid messages to trading partners and internal systems.

The Mapper helps you generate XSLT that generates valid XML, and it warns you about certain issues when validating the map or when compiling the project. Ultimately, however, it is the responsibility of the developer to make sure that the output generated by a map is valid.

This means that after developing your map you want to test it. Preferably, test all possible cases the map can get into at runtime. This section walks you through your options for testing your map.

Validating Maps

First of all, when developing a map, you should validate it. This is done by right-clicking the map file (.BTM) in Solution Explorer and choosing **Validate Map**. This will let Visual Studio 2010 go through the map and check for different kinds of syntactical errors such as functoids with the wrong number of inputs and other such things. If no errors occur, the map can be compiled and deployed.

The validation might be successful but with some warnings. If warnings occur, you must decide whether to ignore them because you know you have handled the issue the warning is about or whether you must do something about it. A warning that you will probably see many times is the "Warning btm1004: The destination node 'NameOfNode' has multiple inputs but none of its ancestors is connected to a looping functoid." Basically, this warning comes because you have multiple source nodes connected to the same output node. This can be by design if you are using multiple Value Mapping functoids to do conditional mapping of values into one node.

TIP

If you want to avoid the warning about multiple inputs to one destination node, you can use a String Concatenate functoid to concatenate the output of your multiple Value Mapping functoids and connect that to the output node.

Warnings are there for a reason, so take them seriously and deal with them all. As a general rule, BizTalk by default does not validate any messages sent out, meaning that your map really needs to be working well.

TIP

When you validate a map, the output window contains a link to the XSLT that is generated. This XSLT can be useful in determining why your map doesn't work. It can enlighten some of the logic used to build the XSLT and thereby help you build the map using the right functoids for the job.

Testing Maps

After validating your map, you can test it from within Visual Studio 2010. To test the map, you need to provide Visual Studio 2010 with a test instance. You can set some properties on a map file (.BTM) in Solution Explorer to facilitate this. Figure 3.40 shows these, and they are explained in Table 3.15.

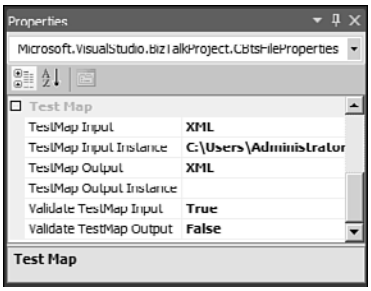

FIGURE 3.40 The properties you can set on a .BTM file in Solution Explorer.

TABLE 3.15 Properties on .BTM Files for Testing a Map

Property	Description
TestMap Input	Can be either Generate Instance, XML, or Native. If set to Generate Instance, Visual Studio 2010 generates an instance of the source schema for the map and uses that as test instance no matter what the other properties are set to. If set to XML, Visual Studio 2010 assumes that the instance you are providing for testing the map is in XML format. If set to Native, Visual Studio 2010 assumes that the instance you are providing is in the native format of the source schema. This allows you to use a flat file or EDI instance as a test instance. Visual Studio 2010 then first converts it into XML using the appropriate schema and editor extensions and uses the XML as input.
TestMap Input Instance	Full path to the file to use as test instance for the map. If the TestMap Input is set to Generate Instance, this property is ignored.
TestMap Output	Can be set to either XML or Native. Determines whether the output of testing the map should be in XML format or the format that is native for the destination schema in the map.
TestMap Output Instance	Full path to where Visual Studio 2010 should write the output of testing the map. If this is not specified, the output is written to a temporary file. The full path to the file that is generated is always written in the output window, giving you access to open it after the test is done.
Validate TestMap Input	If set to true, the instance that is either generated or read from a file is validated against the schema before the map is executed. If set to false, the map is executed without validation.
Validate TestMap Output	If set to true, the output of the map is validated against the schema for the output. If set to False, the output is not validated and written to a file as is.

TIP

When developing a map, it is useful to set the value of Validate TestMap Output to False until you are quite sure the map is working. This is because it allows you to test your map even though it isn't finished. This way you can build the part of the map that creates a part of the destination document and test that before starting on the rest of the map. When satisfied, you can enable the validation and make sure the validation succeeds.

After setting the properties as you want, you can test the map by right-clicking the .BTM file and choosing **Test Map**, as shown in Figure 3.41.

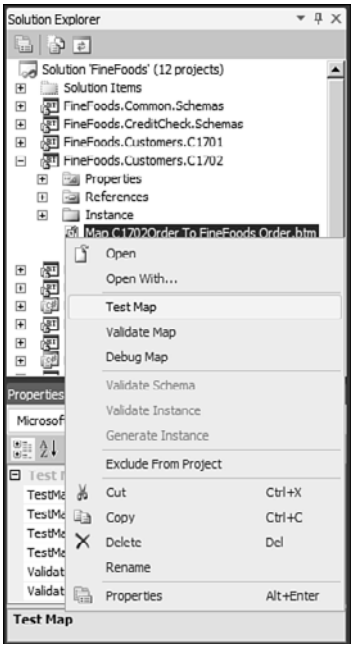

FIGURE 3.41 How to test your map.

After you choose the Test Map option, Visual Studio 2010 reads the properties on the .BTM file as specified in Table 3.15 and tests the map. If the test is successful, you get a link to the generated output in the output window. If the test fails, you receive a list of errors in the Error List window. A test is considered successful if no exceptions are thrown during execution and if input and output validation succeeds, if turned on. Exceptions can occur during execution if the Assert functoid is used or if a functoid actively throws an exception.

Debugging a Map

If your map does not provide you with the output you need and expect, some debugging might be in order. BizTalk supplies you with the option to debug your map line for line to see what happens.

Unfortunately, this functionality does not work for referenced functoids because the debugger cannot find the right external assemblies at runtime. If you want to debug your map and you are using functoids, it is therefore a good idea to make sure you are using the inline versions of all functoids, where possible. The functoids that are shipped with BizTalk, for instance, often have both an inline implementation and a referenced implementation. Which implementation to use when a functoid supports multiple implementations is controlled by the Script Type Precedence property of the Mapper grid. When you click the ellipsis for that property, you get a small window where you can set the script type precedence, as shown in Figure 3.42.

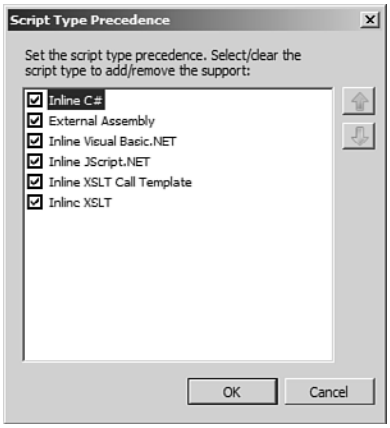

FIGURE 3.42 Setting the Script Type Precedence property.

Basically, you should get the External Assembly possibility moved to the bottom to make sure external assemblies are used as a last resort. If you trust that the inline versions of the functoids is equal to the version that is referenced, you can change the order back after debugging if you want to.

After setting the Script Type Precedence, you need to set the value of the TestMap Input Instance property to point to the instance you want to use as input when debugging the map. You can also specify the TestMap Output Instance if you would like to control the filename the output is written to.

After setting these properties, you can right-click your .BTM file in Solution Explorer and choose **Debug Map**. Visual Studio 2010 then generates the XSLT that is to be debugged and opens it with a breakpoint already set on the first line. Besides this pane, two other panes are also opened. The first contains the output, which lets you keep track of the output that is built while the map is being debugged, and the other is the input XML,

which lets you see which fields in the input XML are currently used to build the output. You can drag the windows around so that you can see all three panes at the same time. Figure 3.43 illustrates this.

FIGURE 3.43 Debugging a map.

As you can see in Figure 3.43, three panes are open: the XSLT, the input XML, and the generated output. As you can also see at the bottom of the figure, you get the watches, as well, thus enabling you to keep track of the values of the variables and optionally change the values at debug time to test what will happen. Also, you can set breakpoints and use **F5** (to run until next breakpoint), **F10** (to step over the currently active line of code), and **F11** (to step into the currently active line of code), as you are used to doing when debugging .NET code inside Visual Studio 2010.

Unit Testing

After a map is developed, validated, tested, and possibly debugged, you should enable unit testing of the map. As a matter of fact, many great people insist you should write your unit tests before even starting developing anything. This is called *test-driven development* (TDD). In either case, BizTalk ships with the option to do unit testing on your maps, which you should leverage, and this section describes this functionality.

The first thing to do is to enable unit testing on the project that contains a map you want to unit test. To do so, follow these steps:

1. Go to Solution Explorer.
2. Right-click the project that contains the map that is to be unit tested, and choose **Properties**.
3. Go to the Deployment pane and enable the **Enable Unit Testing** property, as shown in Figure 3.44.

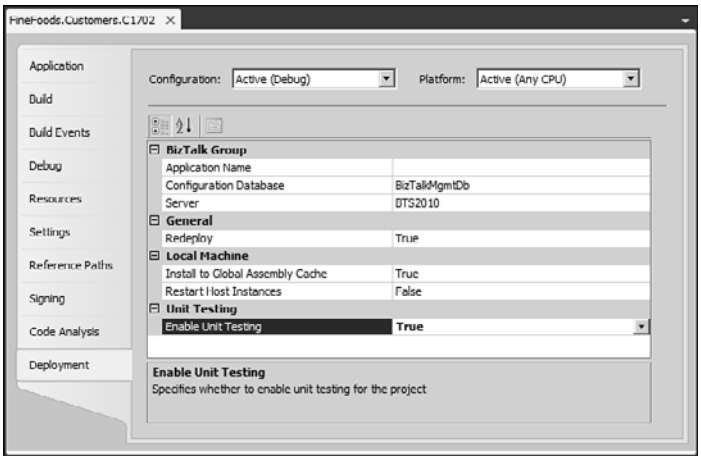

FIGURE 3.44 Enabling unit testing on project.

After enabling unit testing on the project, all maps in the project that are compiled will be inheriting from the `TestableMapBase` class in the `Microsoft.BizTalk.TestTools.Mapper` namespace instead of the normal `TransformBase` class in the `Microsoft.XLANGs.BaseTypes` namespace. The `TestableMapBase` class actually inherits from `TransformBase`, so nothing is lost. What is gained, however, are some methods and properties that can be leveraged for unit testing.

Next, you should add a test project to your solution. This is done by using the menu in Visual Studio 2010, where you can click **Test**, **New Test** to open the screen shown in Figure 3.45. In this screen, you can choose to either add the new test to an existing test project, if one is present in the solution, or create a new Visual C# test project. In addition, you can choose between four different tests:

- **Ordered Test:** This gives you a way of orchestrating your tests, deciding what order they should be performed in.
- **Unit Test:** This option gives you a class to write your tests in, but some manual work needs to be done like referencing the right assemblies and so on.
- **Basic Unit Test:** This option provides an even smaller and simpler version of a unit test class than the unit test. More to implement yourself.
- **Unit Test Wizard:** This helps you through some of the choices you must make, like what maps to test, and then generates the test class for you.

The other five options for adding a new test are not relevant for a BizTalk project.

For your first test project, name your file, choose **Unit Test**, and click **OK**. If you already have a test project, you can decide to add the file to an existing project by selecting it in the drop-down.

The test project is created for you, and it includes a pretty empty class for your tests. The class contains some definitions, methods, a constructor, and one method that is really the one thing to focus on, because this is the one you need to implement.

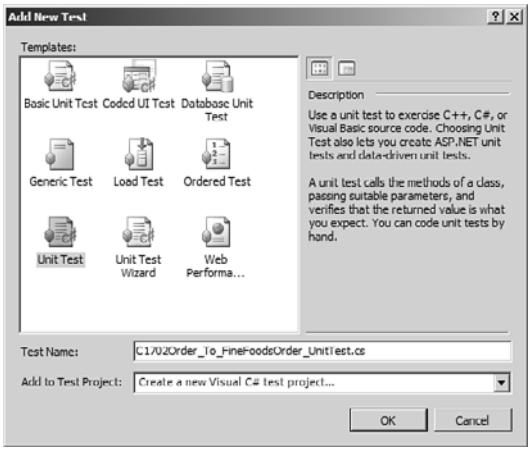

FIGURE 3.45 Adding a new test to your test project.

So, what you need to do in this method is to implement the unit test of the maps you want to test in this test class. Remember that you can have as many test classes and test methods in each test project as you want.

To test a map, you need to reference three different assemblies from your test project:

- **The assembly that contains the map:** This assembly *must* have the Enable Unit Testing property set to True.
- **The Microsoft.BizTalk.TestTools assembly:** This is found in the .NET pane of the Add Reference screen.
- **The Microsoft XLANG/s Base Types assembly:** This is also found in the .NET pane of the Add Reference screen.

In the project, the XML instances that are to be used for the unit test can also be added. If you do this, consider marking their Build Action property to None, so they are not compiled needlessly into the assembly. The instances are not required to be included in the project, but it gives you a nice way of having everything needed for the test grouped together.

After adding the project references, you need to implement the test method. Locate the method in the class file called TestMethod1, and change it to something like the code shown in Listing 3.20.

LISTING 3.20 Sample Test Method for Testing a Map

```
[TestMethod]
public void TestMapC1702OrderToFineFoodsOrder()
{
    string INPUT = testContextInstance.TestDir + @"\..\Order.xml";
    string OUTPUT = testContextInstance.TestDir + @"\..\MappedOrder.xml";
    TestableMapBase map = new Order_to_InternalOrder();
    map.ValidateInput = true;
    map.ValidateOutput = true;
    map.TestMap(INPUT, InputInstanceType.Xml,
        OUTPUT, OutputInstanceType.XML);
    Assert.IsTrue(File.Exists(OUTPUT), "File does not exist");
    // Read in OUTPUT and check relevant values.
    // Compare file with expected output.
}
```


The two strings `INPUT` and `OUTPUT` are declared to contain the path to the input instance for the map and the path to the output file to write the output to. The functionality required basically instantiates the map as a `TestableMapBase` class, which contains the needed properties and methods for unit testing. Then the properties `ValidateInput` and `ValidateOutput` are set to `true`. These properties mean the same as the properties you can set on the .BTM file and will determine whether the `TestMap` method should validate the input and output against the respective schemas before and after the map is executed. Any failures in this validation results in the test failing. Both values are `false` by default.

For the code to compile, the `using` statements shown in Listing 3.21 are needed. The namespaces are as follows:

- `Microsoft.VisualStudio.TestTools.UnitTesting`: This is the namespace needed to use the `TestClass` and `TestMethod` attributes.
- `FineFoods.Customers.FamilyRestaurant`: This is the namespace the map is located in, providing you access to the `Order_to_InternalOrder` class that is the compiled map.
- `Microsoft.BizTalk.TestTools.Mapper`: This namespace contains the `TestableMapBase` class needed to call methods on the map object.
- `Microsoft.BizTalk.TestTools.Schema`: This namespace contains the two enumerations used for specifying the type of input and the type of the output for the `TestMap` method.
- `System.IO`: This is used to be able to access the `File` class that is used to check whether the output file exists in the assertion that is in the `test` method.

LISTING 3.21 **using** Statements Necessary for Code in Listing 3.20

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using FineFoods.Customers.C1702;
using Microsoft.BizTalk.TestTools.Mapper;
using System.IO;

using Microsoft.BizTalk.TestTools.Schema;
```

The code in Listing 3.20 shows just one assertion, which basically asserts that the output file is created. You might need other assertions, such as comparing the output to an instance of the expected output or reading in the output file and validating some of the actual values created.

You should have as many instances as needed to make sure your map can handle all possible instances correctly. This means you should have plenty of input instances and possibly plenty of samples of the output.

You can find Microsoft's description of the unit test features at http://msdn.microsoft.com/en-us/library/dd224279(BTS.70).aspx.

Summary

A map is a transformation from one XML format into another. It is important that maps generate valid output, because otherwise you can send invalid XML to customers, trading partners, internal systems, and others. Because the input to a map can be quite complicated XML with lots of optional fields and even fields that are not optional yet are still marked as such in the schema, developing a single map that deals with the possible inputs to the map can present a big challenge. Take care and study the input and output schemas and determine how to handle all special cases.

Maps are developed in the Mapper, which in turn is converted into XSLT, and you can even provide your own XSLT instead of that which is generated. BizTalk Server 2010 provides an all-new Mapper that helps developers get a good overview of the map to keep track of it and that helps developers change an existing map. The Mapper provides a nice and intuitive user interface for building your transformations. It provides functionality to both copy values directly from the source to the destination and also functoids that provide functionality that is performed on the source before it is mapped to the destination.

BizTalk ships with a lot of functoids that you can use as building blocks for the most common mapping challenges. If you run into either functionality that the built-in functoids cannot handle or a combination of functoids you use a lot and want to group together, you can build your own functoids. Custom functoids have access to the entire .NET framework for referenced functoids and a subset of the .NET framework for inline functoids, providing you with the power of expression you need to do almost anything.

Index

A

B

C

D

E

F

G

H

I-J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z