Andrey Tserkus

Magento for Developers: Product Configuration and “Composite Products” Functionality

Magento knowledge required: Medium

- Detailed description of product configuration management in Magento
- New features implemented within the “Composite Products” story
- Full developer’s guide to internal design implementation
- Checklists for module and theme developers
- Directly from the source – written by the lead of the Magento Core feature group

“The guide is excellent. It has a very practical approach, while the reading is entertaining and easy. I’m a BIG fan!”

Vinai Kopp, Senior Manager of Developer Education at Magento, Inc.

“The guide features a thorough coverage of a topic. And the images together with the diagrams are an excellent complement to the text.”

Tim Reynolds, Magento Architect at Briteskies, LLC

“Reading the guide, we really liked its quality and technical level. A strong point of it is the emphasis on the architectural concept of the described Magento functionality. The guide creates a profound knowledge and leads to better code.”

Jan Mech, Magento Technical Lead at OrderMotion, Inc.

Andrey Tserkus

Magento for Developers:

Product Configuration and “Composite Products” Functionality

First Printing: December 2011

Contents

Preface

My name is Andrey Tserkus and I am a developer in the Magento Core Team.

Our department works on the heart of Magento – the core modules and main architecture of the system – so no one knows the system better than we do.

From time to time I communicate with the Magento community in blogs or forums, tweet on Twitter, answer questions at Stack Overflow or just go to conferences to meet people in person. This guide is one more way to support the whole ecosystem of Magento developers. It's not some kind of official guide, although its creation was supported by Magento, Inc. It's mostly just my personal effort to give more documentation to the community.

I wrote this guide on my own and I checked it for accuracy. In addition, a team of reviewers (professional Magento developers and trainers) from all over the world examined it. After that, a tech-writer completed the process. However, if you find logical inaccuracies or language mistakes, or if you have any suggestions – do not hesitate to drop me a note at andrey.tserkus@magento.com. It will help improve this guide, and possibly future guides as well. You can also reach me on Twitter: @Zerkella

I am really looking forward to receiving your comments on this guide. Thanks and enjoy reading it.

Acknowledgments

I'm really grateful to all the volunteer reviewers who read the guide and gave valuable feedback on it. Among them I would name: Ashley Schroder, Tsvetan Stoychev, Jan Mech, Patrick Puck, Tim Reynolds and Rouven Alexander Rieker. And I would like to thank Elena Velichuk for her advice on diagrams in this guide.

Very special thanks go to Vinai Kopp for his detailed work and many improvements to the text.

I'm also thankful to Yoav Kutner, CTO of Magento, Inc., for his support on the guide preparation process. And the language quality would not be possible without Shannon McIntyre, our technical writer in Magento, Inc., who carefully proof-read the text.

Introduction

Magento has a huge number of features, modules and extensions. But all of its functionality is centered around the most important part of the system – catalog products. They are the main reason for a store owner to build a store, and the main reason for a customer to visit that store. With great diversity among different goods in the world, it is a challenge for Magento to be able to compose the best view of any type of product and allow customers to make custom configurations for the products ordered (i.e. choose color, size, enter labels to print and so on).

A great deed has been done by Magento developers recently in extending product configuration management within the platform. With the Magento feature release CE 1.5 / EE 1.10, a new set of yummy features was presented. Because the community has a strong demand for documentation, I have composed this useful developer's guide for that set of features, called **Composite Products**. This is a feature that I led the development process on, so this guide gives a good description of the implemented solutions and the ways for a Magento developer to work with the newly-introduced functionality.

This guide will remain relevant and highly useful for all subsequent Magento releases, because it describes the internal structure of Magento, the Core team's design intentions, and some best practices for interacting with the system. If you're a professional Magento developer or store owner, then this guide is definitely a “must read” for you.

This guide gives a detailed explanation of the Composite Products functionality and covers:

- Complete information about general product configuration and manipulation in Magento
- The ability of a shopper to reconfigure products added to the Shopping Cart or Wishlist
- The improved Admin Panel (store owner/administrator's interface), used to set up product configurations during order creation and customer account management, and the AJAX-architecture on these pages
- Changes made to models to support these features
- Template and layout modifications, required to incorporate Composite Products functionality into themes
- New concepts presented in Magento 1.5, best practices and solutions architecture

This guide is written mainly for Magento developers. However, the first part (till I get to the implementation details) is interesting for store owners too, as it describes new features and the business logic of the system. This text is not suitable for Magento beginners – you should at least be on an intermediate level and have some experience working with the platform.

Composite Products – General Information

First of all, what does the term “Composite Products” mean?

Composite Products is a name for the collection of already existing product types that allow a shopper to configure a product before buying it.

Remember these words – they describe the term completely, and no other words are needed. The Composite Product is neither a new product type, nor a special attribute of a Catalog Product. It's a way of categorizing already existing product types and a codename for some new functionality implemented in Magento 1.5.

In this guide I refer to Magento CE 1.5 and EE 1.10 releases as Magento 1.5. Both these releases share a common codebase snapshot from Magento Core. I will make note of additional EE features where necessary.

Consider the following example: when we want to talk about product types that are not physical and will not be shipped, we use the term *Virtual Products* (actually, Virtual and Downloadable product types both fall into that category). The same is true for Composite Products; when we want to talk about product types that give shoppers the ability to configure them before purchasing, we use the term *Composite Products*.

The whole concept of improving the manipulation of product configurations across the system in Magento 1.5 is referred to as *Composite Products functionality*. You can see it mentioned in release notes and in Magento articles. The Composite Products functionality was implemented for shoppers who configure products before adding them to the cart, and for store managers who configure items while completing orders at the backend.

Do not confuse the term “Composite Products” in this guide with the existing `isComposite()` indicator method in the Product model. They are not related. `isComposite()` means that a product is a container for other products. Alternatively, when I say “Composite Product” in this guide, I mean “a product that has configurable options”.

Although Composite Products is a specific feature improvement developed for a specific Magento release, it is also part of the general product configuration management in Magento. These concepts are inseparable, thus product configuration management is also explained in this guide.

Composite Products in Magento

What types of Magento products are Composite Products? It's easy to answer this question just by checking each product type's functionality and finding the types that can display configuration controls on the Catalog View Product page.

Don't spend your time – I've already composed the list of composite products for you. They are:

- Any product with Custom Options
- Grouped Products
- Configurable Products
- Downloadable Products (with links that can be purchased separately)
- Bundle Products
- Giftcards (Magento Enterprise Edition)

All these product types have been affected by changes made inside the Magento core modules during the Composite Products implementation.

The Purpose of Composite Products

What's the purpose of the Composite Products functionality?

The purpose is very clear. Just take 10 seconds and try to answer the question yourself. 10, 9, 8... 2, 1, 0.

PROFIT!

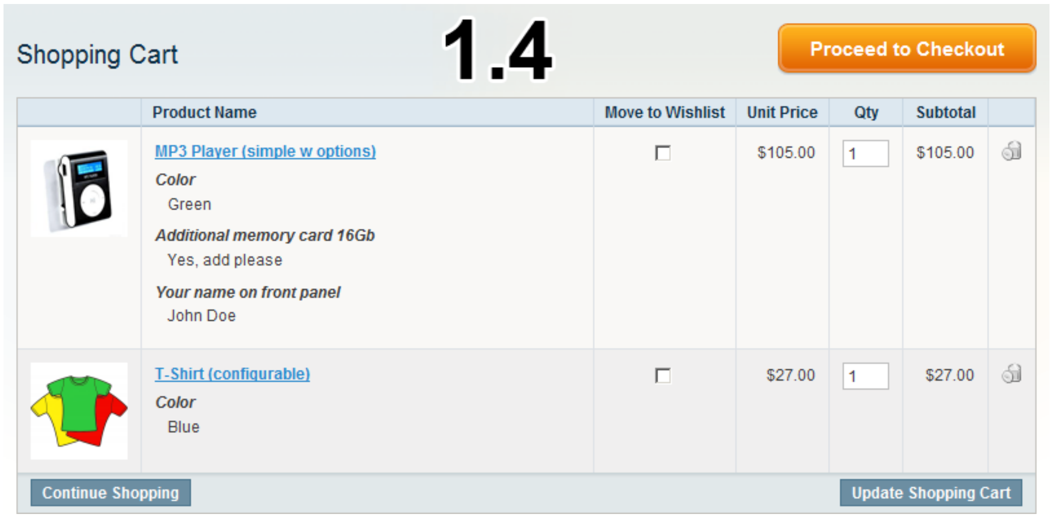
To help store owners achieve this purpose, Magento improved its capabilities for working with product configurations. The changes are big, but their list is small (and I like it for being so clear). Beginning with Magento 1.5:

- Shoppers can easily reconfigure products in both the Shopping Cart and the Wishlist.
- Products are stored in the Wishlist along with their configurations, so they can later be transferred to the Shopping Cart with a single click.
- Store Owners and Admins can create orders in the Admin Panel for all product types in Magento, and they can easily configure products visually because they have the same controls as shoppers in the frontend. This is very useful for call centers. Well, for regular stores it's very convenient, too.
- The *Manage Customer* page in the Admin Panel now allows store owners and admins to view and edit product configurations in shoppers' Shopping Carts and Wishlists.
- The *Manage Shopping Cart* backend page (Magento EE) now allows Store Owners and Admins to view or edit product configurations in shoppers' Shopping Carts or Wishlists. Also, it's possible to configure products and add them to the Shopping Cart from the Catalog, Wishlist, or Comparison List, as well as the Recently Compared, Recently Viewed and Recently Ordered Products lists.

Visual Changes in Magento CE 1.5 / EE 1.10

To make things even clearer, let me show you the visible differences between Magento 1.4 and 1.5.

As you see in Fig. 2, the Shopping Cart page now shows an *Edit* link for each product in the cart.

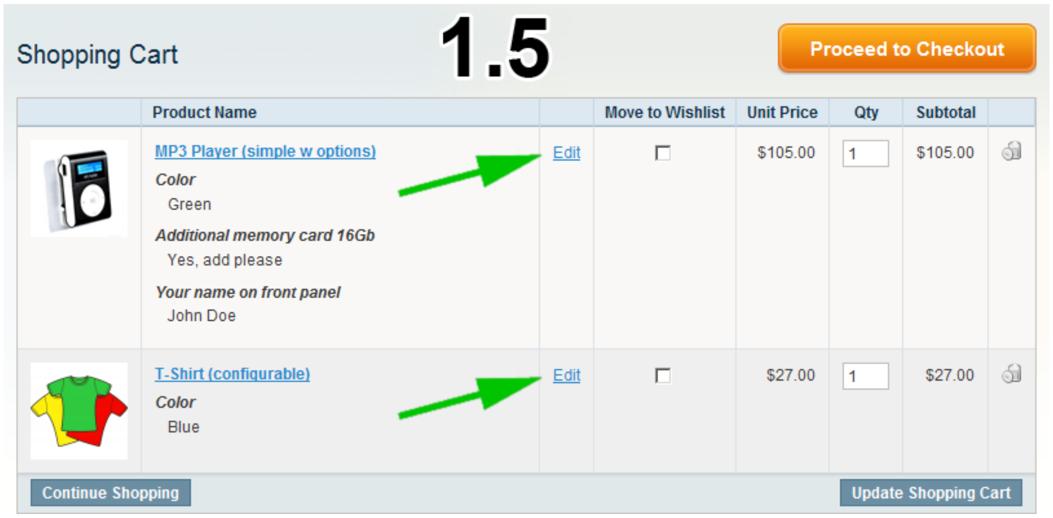


Shopping Cart

1.4

	Product Name	Move to Wishlist	Unit Price	Qty	Subtotal	
	MP3 Player (simple w options) *Color* Green *Additional memory card 16Gb* Yes, add please *Your name on front panel* John Doe	☐	$105.00	1	$105.00	
	T-Shirt (configurable) *Color* Blue	☐	$27.00	1	$27.00	

Fig. 1. The Shopping Cart page in Magento 1.4



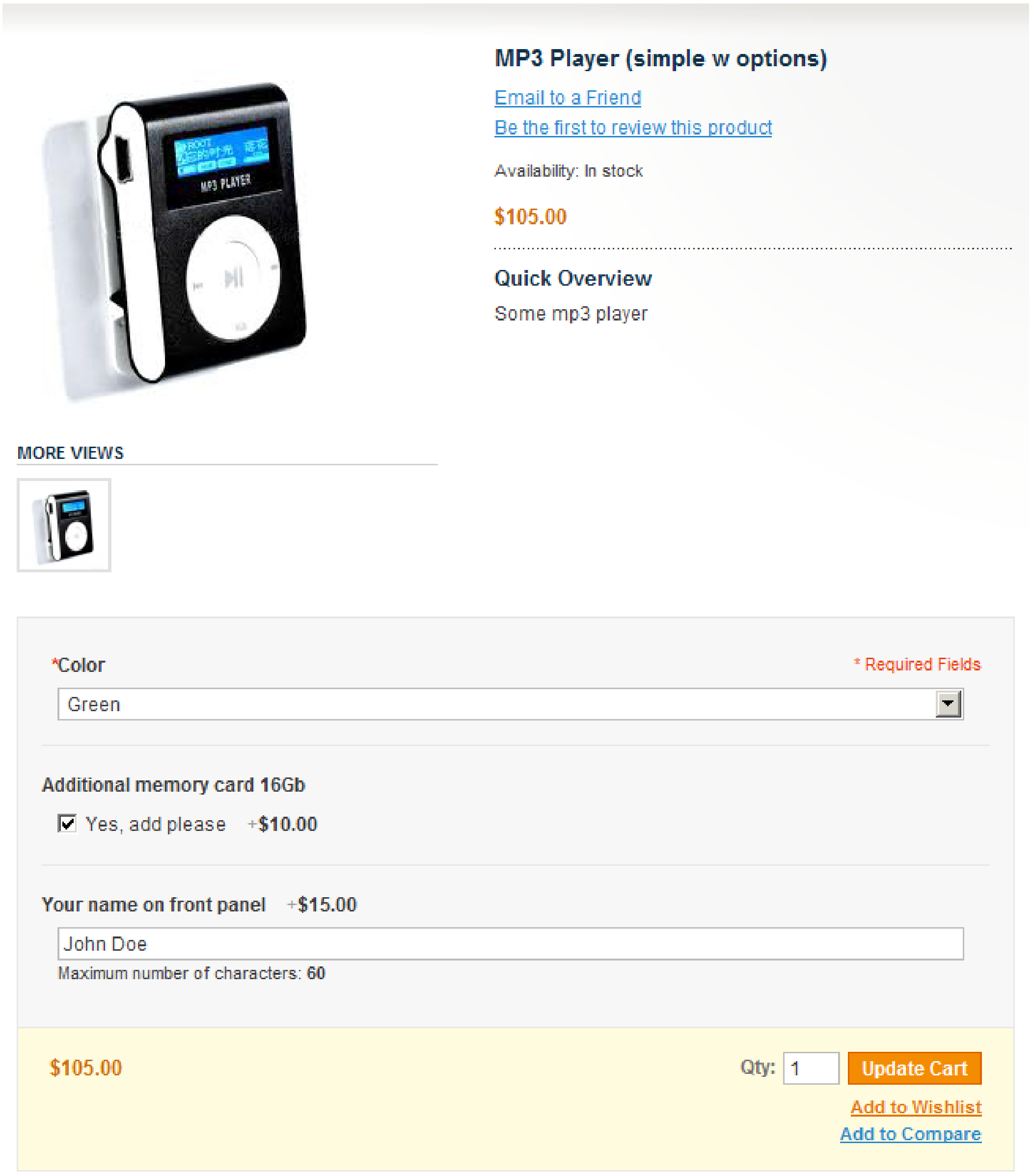
Shopping Cart

1.5

	Product Name		Move to Wishlist	Unit Price	Qty	Subtotal	
	MP3 Player (simple w options) *Color* Green *Additional memory card 16Gb* Yes, add please *Your name on front panel* John Doe	Edit	☐	$105.00	1	$105.00	
	T-Shirt (configurable) *Color* Blue	Edit	☐	$27.00	1	$27.00	

Fig. 2. The Shopping Cart page in Magento 1.5

Clicking the Edit link takes the shopper to the View Product page (Fig. 3), where the controls display currently selected values. Changes made here update the product in the Shopping Cart.



MP3 Player (simple w options)

Email to a Friend
Be the first to review this product

Availability: In stock

$105.00

Quick Overview

Some mp3 player

MORE VIEWS

*Color — * Required Fields

Green

Additional memory card 16Gb

☑ Yes, add please +$10.00

Your name on front panel +$15.00

John Doe

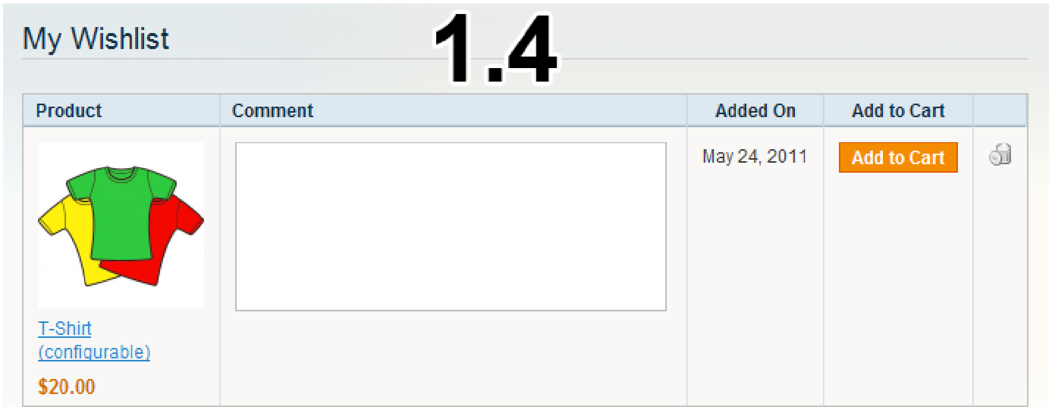
Maximum number of characters: 60

$105.00 — Qty: 1 — Update Cart

Add to Wishlist
Add to Compare

Fig. 3. Editing product configuration in Magento 1.5

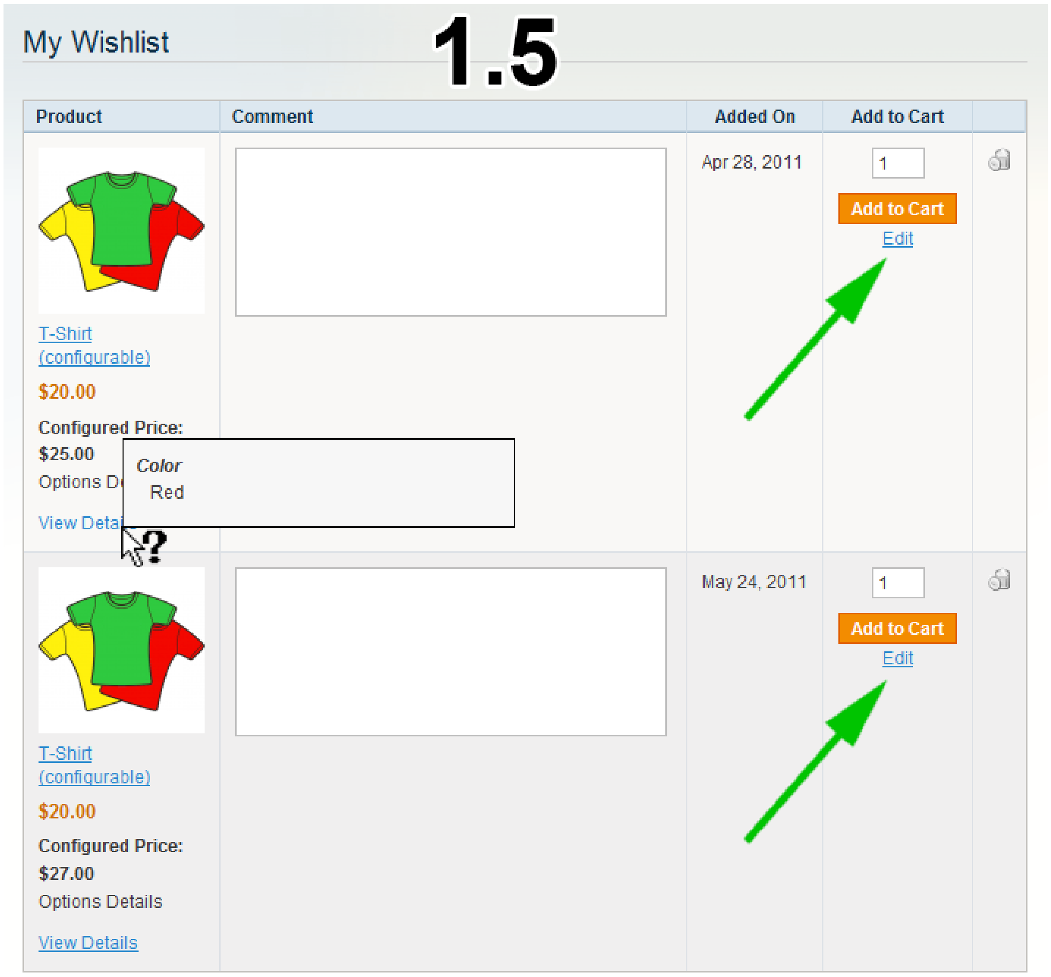
The Wishlist got the Edit link too (see Fig. 5), and it works the same way as the Edit link in the Shopping Cart. The Wishlist also shows the details of the product configuration and allows shoppers to store multiple configurations of the same product.



My Wishlist

Product	Comment	Added On	Add to Cart	
T-Shirt (configurable) $20.00		May 24, 2011	Add to Cart	

Fig. 4. The Wishlist page in Magento 1.4


My Wishlist

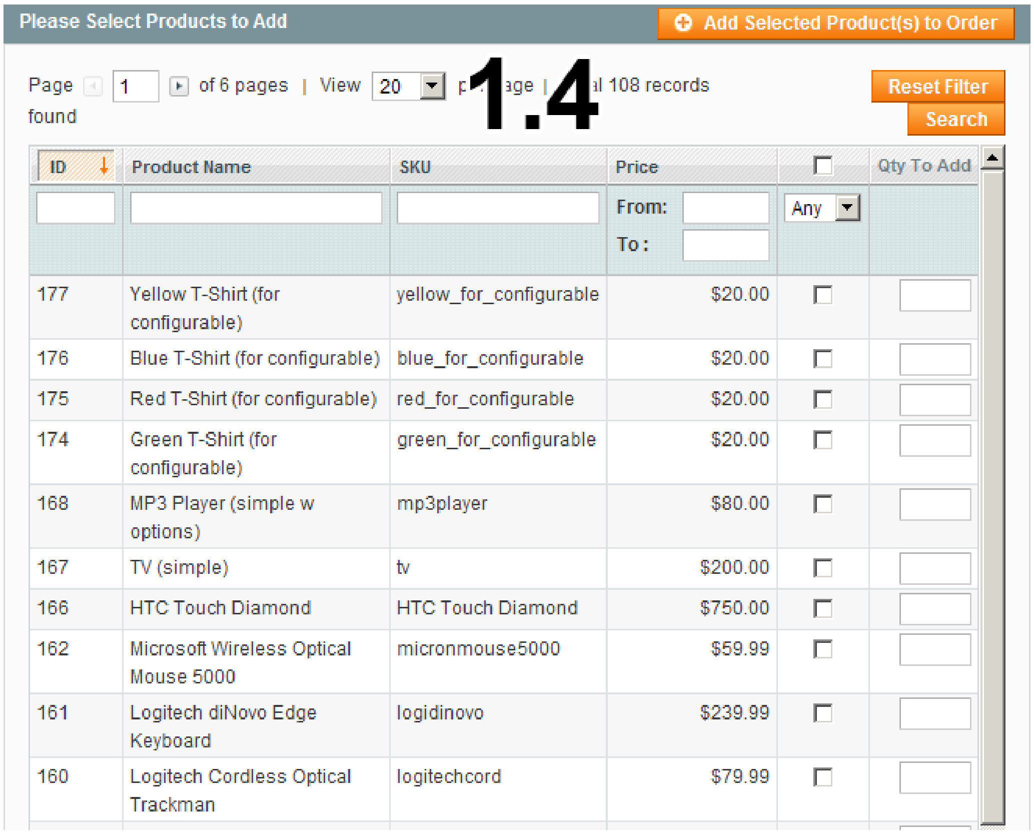
1.5

Product	Comment	Added On	Add to Cart	
T-Shirt (configurable) $20.00 Configured Price: $25.00 Options Details View Details		Apr 28, 2011	1 Add to Cart Edit	
T-Shirt (configurable) $20.00 Configured Price: $27.00 Options Details View Details		May 24, 2011	1 Add to Cart Edit	

Color
Red

Fig. 5. The Wishlist page in Magento 1.5

Previously, during backend order creation in Magento 1.4, only non-composite products were displayed (Fig. 6). Magento 1.5 allows configuration of all product types (Fig. 7).

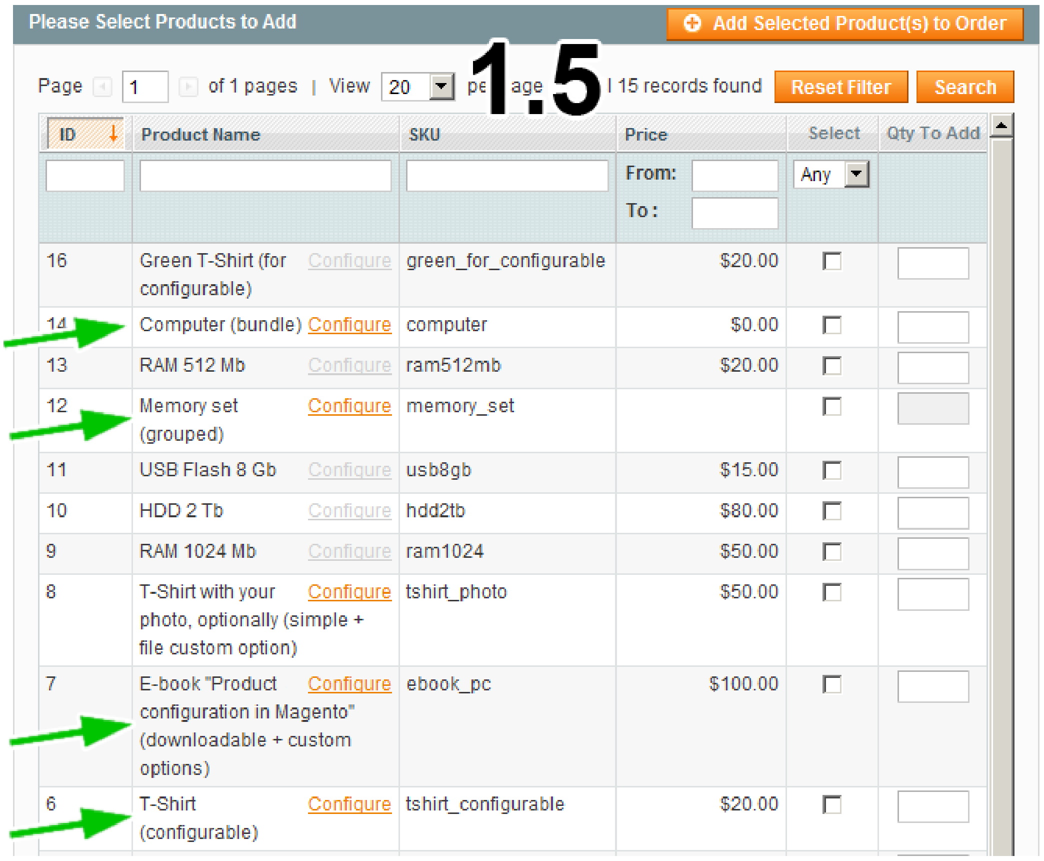


Please Select Products to Add

Page 1 of 6 pages | View 20 p 1.4 age | l 108 records found

ID	Product Name	SKU	Price	☐	Qty To Add
			From: To :	Any	
177	Yellow T-Shirt (for configurable)	yellow_for_configurable	$20.00	☐	
176	Blue T-Shirt (for configurable)	blue_for_configurable	$20.00	☐	
175	Red T-Shirt (for configurable)	red_for_configurable	$20.00	☐	
174	Green T-Shirt (for configurable)	green_for_configurable	$20.00	☐	
168	MP3 Player (simple w options)	mp3player	$80.00	☐	
167	TV (simple)	tv	$200.00	☐	
166	HTC Touch Diamond	HTC Touch Diamond	$750.00	☐	
162	Microsoft Wireless Optical Mouse 5000	micronmouse5000	$59.99	☐	
161	Logitech diNovo Edge Keyboard	logidinovo	$239.99	☐	
160	Logitech Cordless Optical Trackman	logitechcord	$79.99	☐	

Fig. 6. The Create Order page in Magento 1.4


Please Select Products to Add

Add Selected Product(s) to Order

1.5

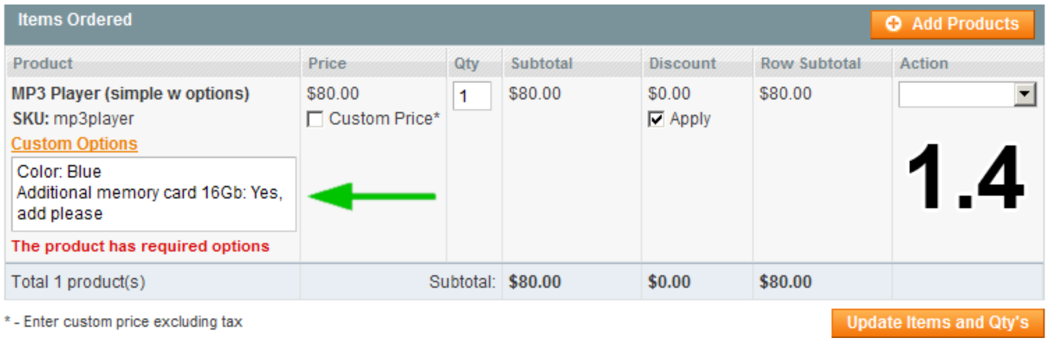
Page 1 of 1 pages | View 20 per page | 15 records found

Reset Filter Search

ID	Product Name		SKU	Price	Select	Qty To Add
				From: To:	Any	
16	Green T-Shirt (for configurable)	Configure	green_for_configurable	$20.00	☐	
14	Computer (bundle)	Configure	computer	$0.00	☐	
13	RAM 512 Mb	Configure	ram512mb	$20.00	☐	
12	Memory set (grouped)	Configure	memory_set		☐	
11	USB Flash 8 Gb	Configure	usb8gb	$15.00	☐	
10	HDD 2 Tb	Configure	hdd2tb	$80.00	☐	
9	RAM 1024 Mb	Configure	ram1024	$50.00	☐	
8	T-Shirt with your photo, optionally (simple + file custom option)	Configure	tshirt_photo	$50.00	☐	
7	E-book "Product configuration in Magento" (downloadable + custom options)	Configure	ebook_pc	$100.00	☐	
6	T-Shirt (configurable)	Configure	tshirt_configurable	$20.00	☐	

Fig. 7. The Create Order page in Magento 1.5

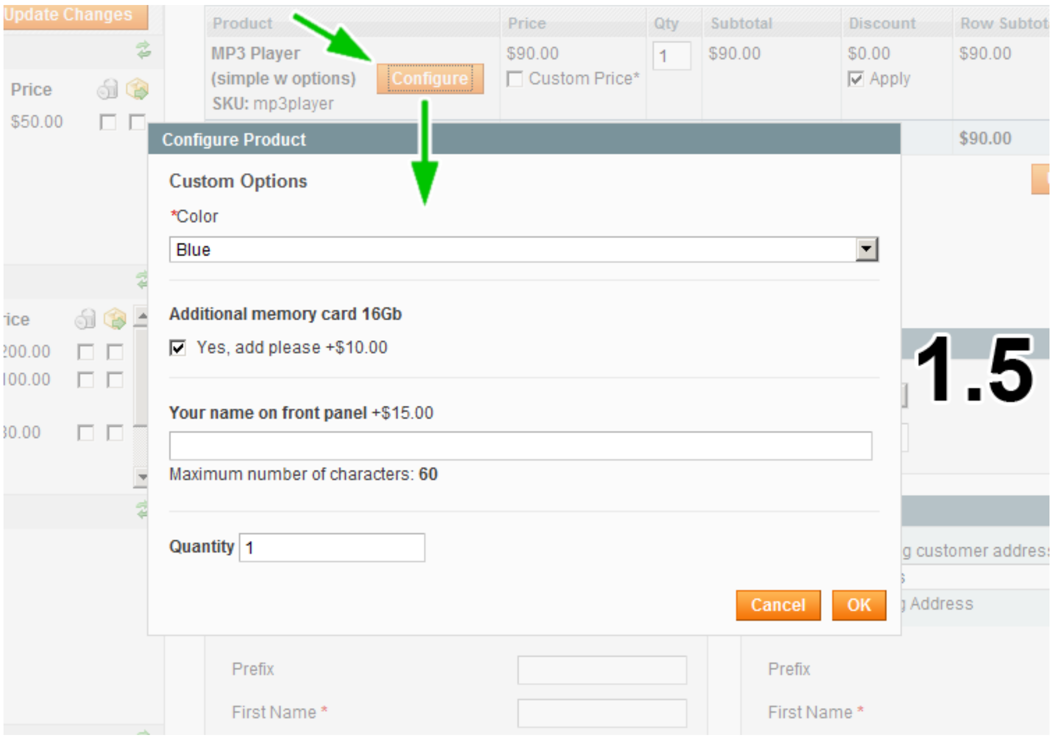
Also, when preparing an order in Magento 1.4, the manager configured the product by entering parameters “hack-style” in a text area control (Fig. 8). In Magento 1.5, all product types have a *Configure* button and are configured via a convenient popup (Fig. 9).



Product	Price	Qty	Subtotal	Discount	Row Subtotal	Action
MP3 Player (simple w options) **SKU:** mp3player Custom Options: Color: Blue Additional memory card 16Gb: Yes, add please — The product has required options	$80.00 ☐ Custom Price*	1	$80.00	$0.00 ☑ Apply	$80.00	
Total 1 product(s)	Subtotal:		**$80.00**	**$0.00**	**$80.00**	

* - Enter custom price excluding tax

Fig. 8. Configuring product in Magento 1.4



Configure Product

Custom Options

*Color

Blue

Additional memory card 16Gb

☑ Yes, add please +$10.00

Your name on front panel +$15.00

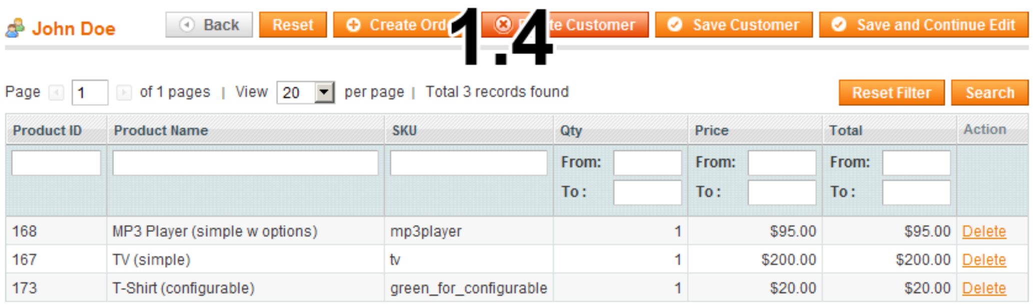
Maximum number of characters: **60**

Quantity 1

Cancel OK

Fig. 9. Configuring product in Magento 1.5

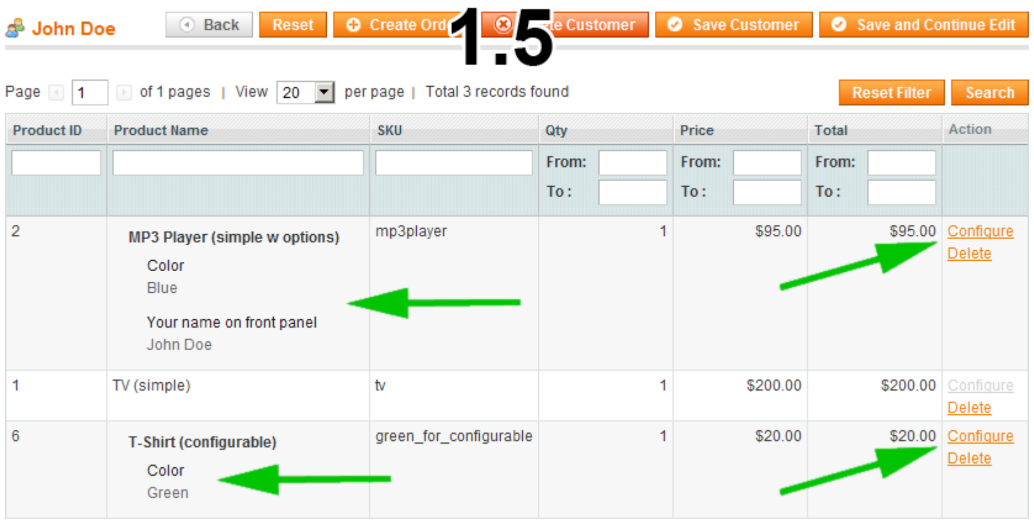
While managing a Customer Shopping Cart or Wishlist in the Magento 1.4 backend, a store manager could only delete items from these lists (Fig. 10). In Magento 1.5, the store manager has two more abilities – to view product configurations and to reconfigure them instantly (Fig. 11).



Page 1 of 1 pages | View 20 per page | Total 3 records found

Product ID	Product Name	SKU	Qty	Price	Total	Action
			From: To :	From: To :	From: To :	
168	MP3 Player (simple w options)	mp3player	1	$95.00	$95.00	Delete
167	TV (simple)	tv	1	$200.00	$200.00	Delete
173	T-Shirt (configurable)	green_for_configurable	1	$20.00	$20.00	Delete

Fig. 10. Customer Shopping Cart view of backend in Magento 1.4



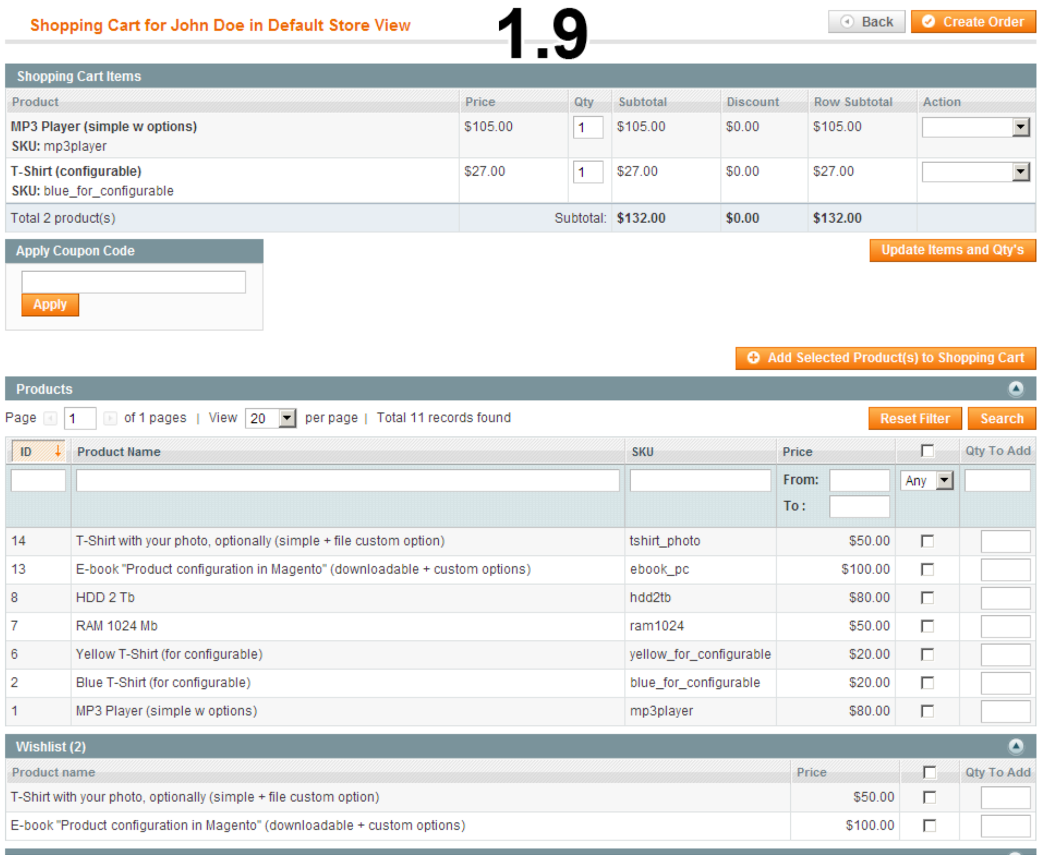
Page 1 of 1 pages | View 20 per page | Total 3 records found

Product ID	Product Name	SKU	Qty	Price	Total	Action
			From: To :	From: To :	From: To :	
2	**MP3 Player (simple w options)** Color Blue Your name on front panel John Doe	mp3player	1	$95.00	$95.00	Configure Delete
1	TV (simple)	tv	1	$200.00	$200.00	Configure Delete
6	**T-Shirt (configurable)** Color Green	green_for_configurable	1	$20.00	$20.00	Configure Delete

Fig. 11. Customer Shopping Cart view of backend in Magento 1.5

Magento EE has the additional ability for store managers to assemble Customer Shopping Carts. The Manage Shopping Cart page in the backend is similar to the Create Order page, as it allows products to be added to the Shopping Cart from the Catalog, Wishlist, Recently Viewed Products, etc., and it also allows coupons to be applied. The Shopping Cart can be converted into an order with the click of a single button.

In Magento EE 1.9 only those product types without configuration options could be added to the Shopping Cart from this page and there was no ability to configure products (Fig. 12). In Magento EE 1.10 any product type can be added to a Shopping Cart, and the page features convenient links and buttons to call the AJAX popup, which provides the ability to configure/reconfigure products instantly (Fig. 13).



Shopping Cart for John Doe in Default Store View

Shopping Cart Items

Product	Price	Qty	Subtotal	Discount	Row Subtotal	Action
MP3 Player (simple w options) **SKU:** mp3player	$105.00	1	$105.00	$0.00	$105.00	
T-Shirt (configurable) **SKU:** blue_for_configurable	$27.00	1	$27.00	$0.00	$27.00	
Total 2 product(s)		Subtotal:	**$132.00**	**$0.00**	**$132.00**	

Apply Coupon Code

Products

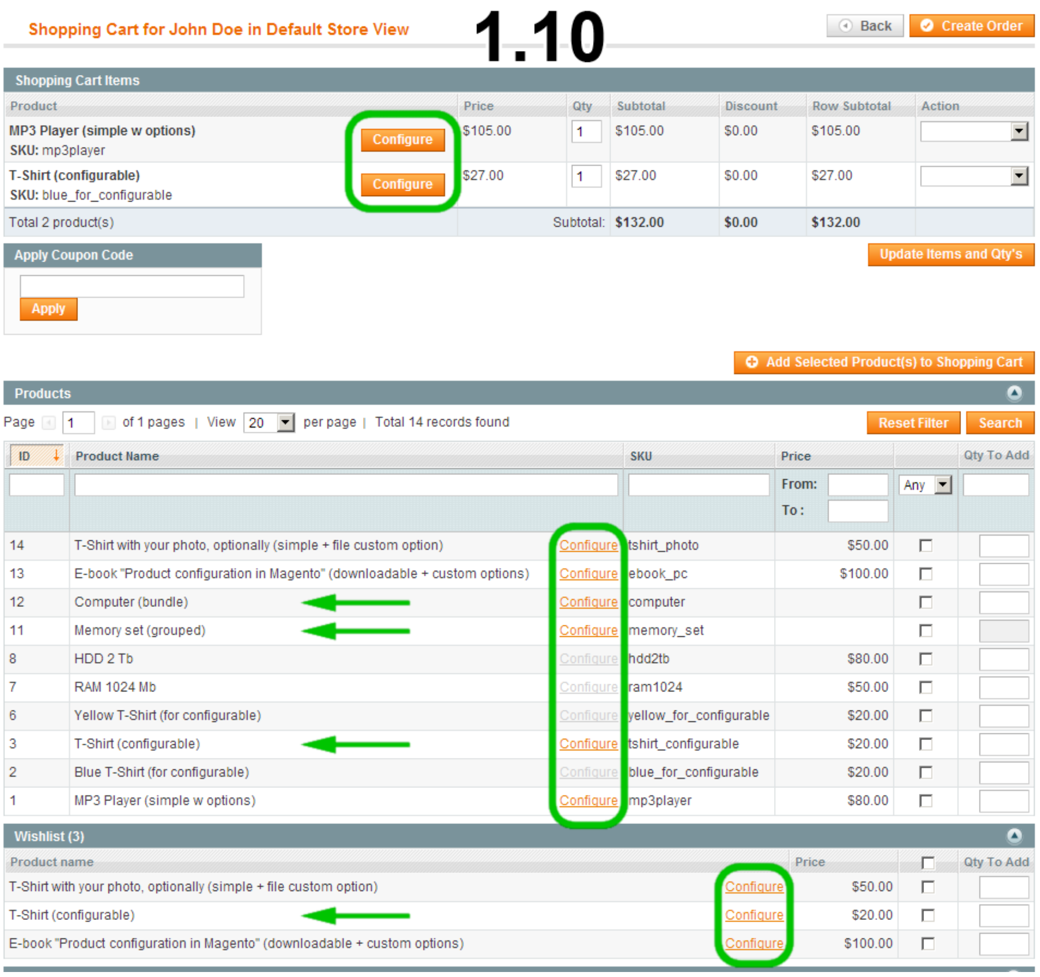
Page 1 of 1 pages | View 20 per page | Total 11 records found

ID	Product Name	SKU	Price		Qty To Add
			From: To:	Any	
14	T-Shirt with your photo, optionally (simple + file custom option)	tshirt_photo	$50.00	☐	
13	E-book "Product configuration in Magento" (downloadable + custom options)	ebook_pc	$100.00	☐	
8	HDD 2 Tb	hdd2tb	$80.00	☐	
7	RAM 1024 Mb	ram1024	$50.00	☐	
6	Yellow T-Shirt (for configurable)	yellow_for_configurable	$20.00	☐	
2	Blue T-Shirt (for configurable)	blue_for_configurable	$20.00	☐	
1	MP3 Player (simple w options)	mp3player	$80.00	☐	

Wishlist (2)

Product name	Price		Qty To Add
T-Shirt with your photo, optionally (simple + file custom option)	$50.00	☐	
E-book "Product configuration in Magento" (downloadable + custom options)	$100.00	☐	

Fig. 12. The Manage Shopping Cart page in Magento EE 1.9



Shopping Cart for John Doe in Default Store View

Shopping Cart Items

Product		Price	Qty	Subtotal	Discount	Row Subtotal	Action
MP3 Player (simple w options) **SKU:** mp3player	Configure	$105.00	1	$105.00	$0.00	$105.00	
T-Shirt (configurable) **SKU:** blue_for_configurable	Configure	$27.00	1	$27.00	$0.00	$27.00	
Total 2 product(s)			Subtotal:	**$132.00**	**$0.00**	**$132.00**	

Apply Coupon Code

Products

Page 1 of 1 pages | View 20 per page | Total 14 records found

ID	Product Name		SKU	Price		Qty To Add
				From: To :	Any	
14	T-Shirt with your photo, optionally (simple + file custom option)	Configure	tshirt_photo	$50.00	☐	
13	E-book "Product configuration in Magento" (downloadable + custom options)	Configure	ebook_pc	$100.00	☐	
12	Computer (bundle)	Configure	computer		☐	
11	Memory set (grouped)	Configure	memory_set		☐	
8	HDD 2 Tb	Configure	hdd2tb	$80.00	☐	
7	RAM 1024 Mb	Configure	ram1024	$50.00	☐	
6	Yellow T-Shirt (for configurable)	Configure	yellow_for_configurable	$20.00	☐	
3	T-Shirt (configurable)	Configure	tshirt_configurable	$20.00	☐	
2	Blue T-Shirt (for configurable)	Configure	blue_for_configurable	$20.00	☐	
1	MP3 Player (simple w options)	Configure	mp3player	$80.00	☐	

Wishlist (3)

Product name		Price	☐	Qty To Add
T-Shirt with your photo, optionally (simple + file custom option)	Configure	$50.00	☐	
T-Shirt (configurable)	Configure	$20.00	☐	
E-book "Product configuration in Magento" (downloadable + custom options)	Configure	$100.00	☐	

Fig. 13. The Manage Shopping Cart page in Magento EE 1.10

There are other visual changes in Magento, but they're much smaller and aren't important enough to discuss in this chapter. I will mention them later.

Business Logic Involved

Before we start working with product configurations, let me tell you some important Magento business rules accentuated in the Composite Products specifications. Knowing them will help you understand what's going on in the system from the shopper's point of view, and how data is processed internally from the developer's point of view.

1. A Wishlist can hold products with full, partial or no configuration.

   When a shopper adds a product to the Wishlist, it's not important to the system for all the options to be set up. It will become important later – during checkout. But for now we let the shopper add a product to the Wishlist in a single click and he isn't bothered with the full product configuration. Just “don't be evil”, as people say :)

2. The product configuration is checked before storing it in Magento.

   We must always make sure that wrong data, huge non-valid files or other hacker's tricks don't make their way into our system. We don't require full product configuration in the Wishlist, but if a shopper does submit something it must be checked for validity. So when adding a product to the Shopping Cart we check that all required product options are set, and that all entered options pass validation. In the case of the Wishlist only specified options are validated.

3. Newly added items are automatically *appended* to items with identical product configurations.

   Whenever a shopper adds a product to the Shopping Cart or Wishlist and the same product (identically configured, e.g. having same color) is already present, the new product is not added as a separate item. Instead, it is appended to the matching product and the quantity of the existing item increases. (see Fig. 14).

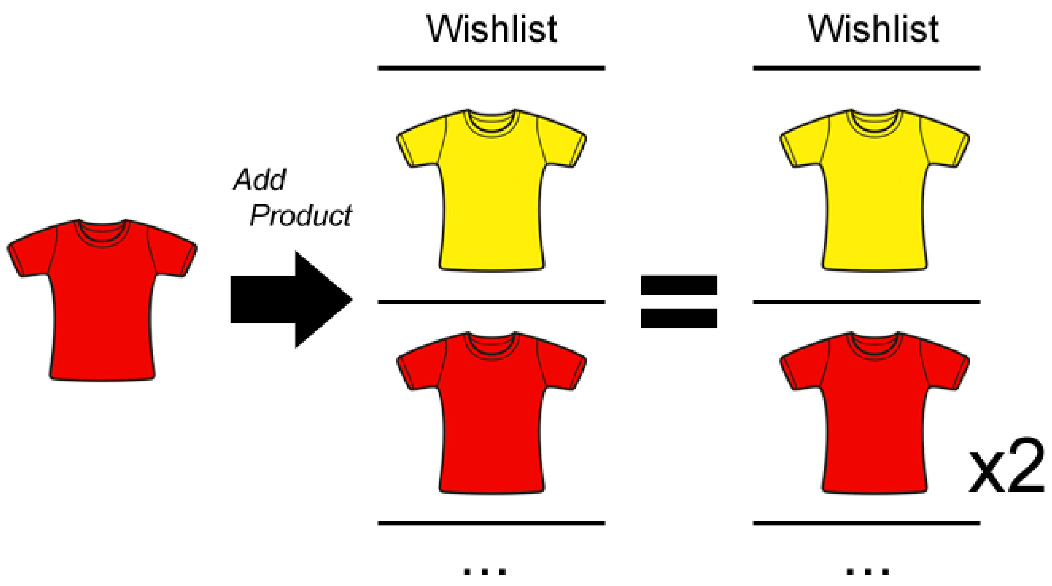


Fig. 14. Appending identical product configurations

Magento doesn't keep a record of creating such a combination. Items, once appended, become a single entity which cannot be separated back into its components. It is a normal item that can be edited as one product (which feels natural for the shopper), and the quantity can be freely changed.

4. Grouped Products are separated into Simple Products in the Shopping Cart. And there is no way back.

   A Grouped Product serves as a quick way to add multiple related products to the Shopping Cart. Previously, Magento added only the Grouped Product's child products to the Shopping Cart. On the other hand, the Wishlist stored the Grouped Product as a single entity. Nothing has changed, really, except the fact that the Wishlist in Magento 1.5 stores Grouped Products with the quantities of the “child” Simple Products and allows those quantities to be modified.

Now that we understand Composite Products from the shopper's point of view, let's move on to the technical description. Store Owners may stop reading here, as the rest of the guide covers programming aspects of the system. Magento developers should continue.

Technical Developments

After walking through the business analysis of the Composite Products functionality, we're ready to start talking about its development internals. Let's think about the things that were available in Magento 1.4, and the things that the Core team needed to implement to fulfill the Composite Products business requirements.

1. The Catalog Product in Magento 1.4 knew how to process incoming POST-requests and create parsed values – *configuration options* – out of them. But Composite Products require a different capability – to parse data not only in **full** mode (when all required options must be set and valid), but also in **lite** mode (when only the entered options are validated) for the Wishlist.
2. The best place to edit the product configuration in Magento 1.4 was the View Product page. It had visual controls, full product descriptions, and the price there was recalculated dynamically when the configuration was changed.

   Due to Magento's convenient layout system, it was easy to start using the same page layout for the "Edit Configuration" action. The things we needed to implement in that mode were:

   - adding the ability to show preconfigured values in controls
   - changing the form action url from "add new product" to "edit product configuration"
   - changing the name of the submit button from "Add to ..." to "Update ..."
3. The View Product page in edit mode generates a POST-request to edit the product configuration. But that request has to be processed – I mean, we needed to develop an *update* action in a controller and an *update()* method for the modified items. These things were added to the Shopping Cart and Wishlist modules in Magento 1.5.
4. Magento 1.4 was able to work with products and their configurations – the Shopping Cart successfully stored configured values. Recurring Profiles and the Gift Registry (EE module) also used the same interface as the Shopping Cart to work with configurations. The Wishlist needed to **gain** this ability, so it was obvious that the Shopping Cart architecture should be used for the Wishlist too.

> The Enterprise Gift Registry module already had functionality similar to what we needed to develop for the Wishlist. Therefore, the Gift Registry was refactored a little in order to make both the Wishlist and the Gift Registry have the same architecture for storing item options. I won't give a separate explanation for the Gift Registry because its item option processing doesn't differ from the Wishlist's. If you're working with Magento EE, then you can read the chapter about the Wishlist and get a full understanding of how the Gift Registry manages its item configurations – it's the same as the Wishlist.

A lot of work needed to be done in the Admin Panel, on the Create Order and Manage Shopping Cart pages. Think about AJAX popups, temporarily storing multiple product configurations in the browser before submitting to the server, the same visual controls used to set up products at the frontend, modification of all the related models' methods in order to accept configurations... ahh, that was a huge piece of work!

Summary

Ok, let's summarize this chapter and make a plan for further discussion on Composite Products implementation.

We already know about things that the Magento Core Team needed to develop:

1. Lite and full configuration parsing modes for Catalog Products.
2. The ability for the Wishlist to work with configurations (using architecture from the Shopping Cart).
3. Editing of product configurations in the frontend of the same page where the product is viewed.
4. *Update* action and *update* method for the controllers and models in the Shopping Cart and Wishlist.
5. Complete backend refactoring, creation of AJAX popups and visual controls for the Admin Panel.

Breaking all those tasks into smaller ones, we come to the mind mapping diagram shown in Fig. 15.

Most of these tasks were explained above, however, some of the smaller subtasks touch deep technical aspects of the system and have not yet been described. We will discuss them later, when the time comes. I have intentionally included all Composite Products concepts into this schema – whether already explained or not. So now we have the full picture for our future discussion, and a useful reference map to move from one place to another.

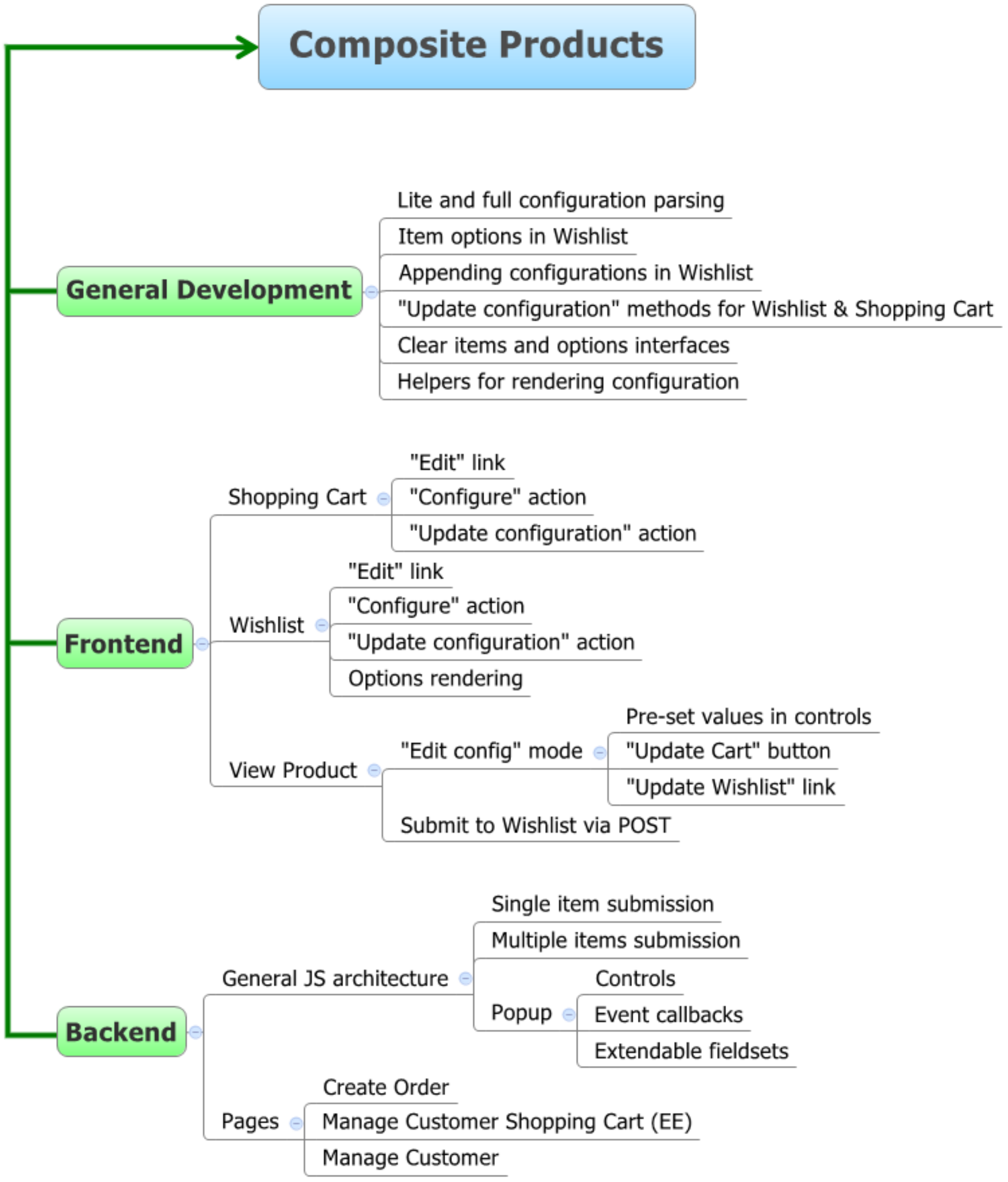

Composite Products
General Development
Lite and full configuration parsing
Item options in Wishlist
Appending configurations in Wishlist
"Update configuration" methods for Wishlist & Shopping Cart
Clear items and options interfaces
Helpers for rendering configuration
Frontend
Shopping Cart
"Edit" link
"Configure" action
"Update configuration" action
Wishlist
"Edit" link
"Configure" action
"Update configuration" action
Options rendering
View Product
"Edit config" mode
Pre-set values in controls
"Update Cart" button
"Update Wishlist" link
Submit to Wishlist via POST
Backend
General JS architecture
Single item submission
Multiple items submission
Popup
Controls
Event callbacks
Extendable fieldsets
Pages
Create Order
Manage Customer Shopping Cart (EE)
Manage Customer

Fig. 15. Structure of the Composite Products Feature

Product Configuration

We have mentioned the term *product configuration* several times. In fact, it is the cornerstone of the Composite Products functionality. But what is it? Where does Magento store it? How is it moved from one item list instance to another (i.e. from the Wishlist to the Shopping Cart)?

The simple definition of the term is:

“Product configuration” is the set of all values entered by a shopper while configuring a product for purchase.

Internally, product configuration is represented by an associative array of item options. The keys in this array are internal codes for the options (e.g. `option_17`, `bundle_selection_12_qty`, etc.). The values are *item options* – lightweight objects holding processed and validated configuration data submitted by a shopper. One item option contains one setting value.

There is also the automatically added option called *buyRequest*. It holds the whole $POST-array with the submitted configuration.

The image in Fig. 16 shows the process flow initiated when a shopper wants to add a product to the Shopping Cart or Wishlist and thus submits a form with the product configuration. Magento receives that form as a POST-request, validates it and parses it into item options. Additionally, the whole POST-request is stored as a separate option with the code `info_buyRequest`. This special option is called a *buyRequest*.

Item options represent the product configuration as a map of [internal option code] to [item option], which contains an actual configuration value selected by the shopper. So *item options* is just an array which holds all user settings for the product – color, size, custom labels, special wrapping and so on.

> The term *item* in this guide means a Shopping Cart item (row), a Wishlist item or any other list item that holds a product with its configuration (e.g., Enterprise Gift Registry item).

Magento uses item options to work with distinct **single** product configuration values. And the buyRequest is used to transfer a product with a **complete** configuration from one list to another (e.g., from the Wishlist to the Shopping Cart). The buyRequest is also needed on the product reconfiguration page.

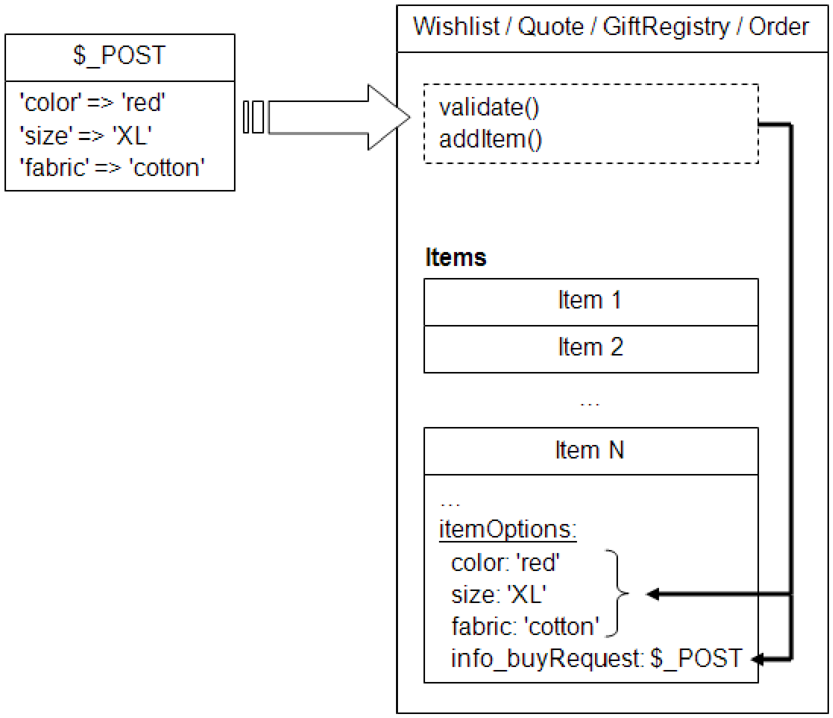


Fig. 16. Item options and the buyRequest

In this section of the guide I am going to explain the use of item options and the role of the buyRequest in integrating one product list with another. Product reconfiguration will be discussed in the following sections of the guide.

Item Options

When working with items, we don't need the buyRequest; item options are usually more convenient. Items in Magento 1.5 expose a common interface to work with their options. We can use a Wishlist item as an example to demonstrate this (Fig. 17).

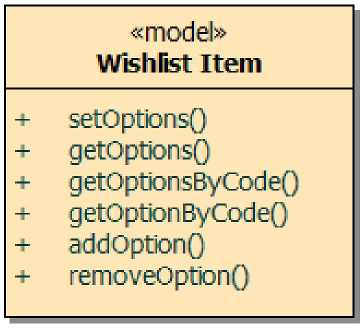


Fig. 17. Wishlist Item methods, designed to work with options

Inside the code these methods are declared in the following way:

```
Mage_Wishlist_Model_Item

/**
 * Initialize item options
 *
 * @param   array $options
 * @return  Mage_Wishlist_Model_Item
 */
public function setOptions($options)

/**
 * Get all item options
 *
 * @return array
 */
public function getOptions()

/**
 * Get all item options as array with codes in array key
 *
 * @return array
 */
public function getOptionsByCode()
```

```
/**
 * Get item option by code
 *
 * @param   string $code
 * @return  Mage_Wishlist_Model_Item_Option | null
 */
public function getOptionByCode($code)

/**
 * Add option to item
 *
 * @param   Mage_Wishlist_Model_Item_Option $option
 * @return  Mage_Wishlist_Model_Item
 */
public function addOption($option)

/**
 * Remove option from item options
 *
 * @param string $code
 * @return Mage_Wishlist_Model_Item
 */
public function removeOption($code)
```

This interface is sufficient to get/set the configuration options entered by the user. It's also used to store special options needed internally for Magento (e.g., it keeps the buyRequest under code 'info_buyRequest').

Let's write an example to see how this API can be used. The code below loads a shopper's Wishlist, scans through all the Wishlist items and outputs the names of the products. For a Configurable Product, the routine also outputs the product configuration – with the name of an actual simple product attached. Remember that the Wishlist may contain items without configuration, so we have an additional “if” for that case.

```
function logCustomerWishlist($customerId, $storeId)
{
    /** @var $wishlist Mage_Wishlist_Model_Wishlist */
    $wishlist = Mage::getModel('wishlist/wishlist')
        ->loadByCustomer($customerId)
        ->setStore($storeId);

    foreach ($wishlist->getItemCollection() as $item) {
        /** @var $item Mage_Wishlist_Model_Item */
        /** @var $product Mage_Catalog_Model_Product */
        echo "---------------\n";
        $product = $item->getProduct();
        echo $product->getName(), "\n";
        if ($product->getTypeId() == Mage_Catalog_Model_Product_
Type_Configurable::TYPE_CODE) {
            $option = $item->getOptionByCode('simple_product');
            if ($option) { // Product configuration
                $subProduct = Mage::getModel('catalog/product')
                    ->setStoreId($storeId)
                    ->load($option->getValue());
                echo "Sub product: " . $subProduct->getName()
                    . "\n";
            } else {
                echo "Sub product: not configured \n";
            }
        }
    }
}

logCustomerWishlist(1, 2); // You should set your actual ids there
```

The code above outputs the following from my demo Magento installation:

```
---------------
TV (simple)
---------------
T-Shirt (configurable)
Sub product: Red T-Shirt (for configurable)
---------------
MP3 Player (simple w options)
---------------
T-Shirt (configurable)
Sub product: not configured
```

Working with item options, we should also be aware that they are shared with the item's product. By *sharing* I mean that the item instance syncs all its options with the product instance (sets the options inside the product). While the item model just needs these options to store the shopper's configuration, the product model needs them to modify the product's behavior. For example, the product's `getPrice()` method checks whether configuration options are set and returns the price according to those options, not just the basic product price.

Option synchronizing only works one way – from item to product. Changes made to the **product options array** (e.g., options added or removed) will not be saved by the item. But the changes made to the **product option's data** (e.g. a new value is set) will be saved by the item, because actual option instances are shared between items and products. Magento recommends using only the item's interface to change options. Configuration options set in the product are intended to be used by the product in a read-only manner.

In a product model these options are called Custom Options. Do not confuse them with the product *custom options* that an admin defines in the backend. Those options are essentially controls that Magento will display on the product's page on the store's frontend. Internally, Custom Option entities have another meaning – in the product model they store the actual configuration values entered by the shopper.

You can use a product's interface (similar to an item's), as shown in Fig. 18, to work with configuration options.

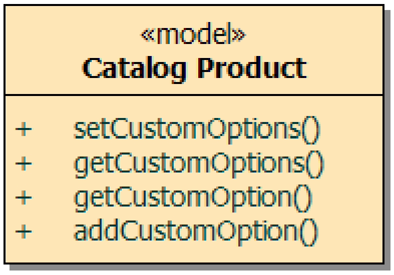


Fig. 18. Product methods, designed to work with its custom options

The internal declaration of these methods looks like:

```
Mage_Catalog_Model_Product

/**
* Set array of custom options to product
*
* @var array $options
* @return void
*/
public function setCustomOptions(array $options)

/**
 * Get all custom options of the product
 *
 * @return array
 */
public function getCustomOptions()

/**
 * Get product custom option info
 *
 * @param   string $code
 * @return  Mage_Catalog_Model_Product_Configuration_Item_Interface
 */
public function getCustomOption($code)
```

```
/**
 * Add custom option information to product
 *
 * @param   string $code
 * @param   mixed $value
 * @param   Mage_Catalog_Model_Product $product
 * @return  Mage_Catalog_Model_Product
 */
public function addCustomOption($code, $value, $product=null)
```

Do you remember the example above, where we logged Configurable Product options? As far as we know, that item's options and that product's options are the same, so we can modify that code to use the product's options instead of the item's – and still get the same output.

```
function logCustomerWishlist($customerId, $storeId)
{
    /** @var $wishlist Mage_Wishlist_Model_Wishlist */
    $wishlist = Mage::getModel('wishlist/wishlist')
        ->loadByCustomer($customerId)
        ->setStore($storeId);

    foreach ($wishlist->getItemCollection() as $item) {
        /** @var $item Mage_Wishlist_Model_Item */
        /** @var $product Mage_Catalog_Model_Product */
        echo "---------------\n";
        $product = $item->getProduct();
        echo $product->getName(), "\n";
        if ($product->getTypeId() == Mage_Catalog_Model_Product_
Type_Configurable::TYPE_CODE) {
            // Previous approach
            // $option = $item->getOptionByCode('simple_product');
            // New approach
            $option = $product->getCustomOption('simple_product');
            if ($option) {
                $subProduct = Mage::getModel('catalog/product')
                    ->setStoreId($storeId)
                    ->load($option->getValue());
                echo "Sub product: " . $subProduct->getName()
                    . "\n";
            } else {
                echo "Sub product: not configured \n";
            }
        }
    }
}
```

```
logCustomerWishlist(1, 2); // You should set your actual ids there
```

The result will look the same as before:

```
---------------
TV (simple)
---------------
T-Shirt (configurable)
Sub product: Red T-Shirt (for configurable)
---------------
MP3 Player (simple w options)
---------------
T-Shirt (configurable)
Sub product: not configured
```

Sometimes, depending on the context, it can be more convenient to use a product instance and not also pass the item to a routine or class which works with the product. In such cases, whenever you need information on a product's configuration, just use the product's custom options. They are synced from the item to the product for exactly such circumstances.

Working with the buyRequest

The buyRequest holds the whole product configuration and is used as a compact record to move a product from one list to another. In fact, it holds a POST-array as it was at the time the product was added to the Shopping Cart or Wishlist.

Magento assumes that if the information in this POST-array was sufficient for a list to understand and parse the values configured by a shopper, then the same array can be used later to add a product with that configuration to another item list. Moving products around the system is done through the same interfaces that are used by shoppers to add/remove products.

Of course, the buyRequest duplicates information stored as item options. Theoretically, these item options could serve to move configurations among lists, but Magento doesn't use item options for this purpose. Lists already have methods to work with data arriving as a POST-array, as it does when a shopper submits a form in the frontend. Why should we create another (duplicating) interface to process the configurations stored in item options? We'd better use existing methods, and provide them with a regular POST-array.

Take a look at the code executed in Magento 1.5 when a shopper moves a product from the Wishlist to the Shopping Cart.

```
Mage_Wishlist_Model_Item

/**
 * Add or Move item product to shopping cart
 *
 * Return true if product was successful added or exception with
 * code.
 * Return false for disabled or invisible products.
 *
 * @throws Mage_Core_Exception
 * @param Mage_Checkout_Model_Cart $cart
 * @param bool $delete  delete the item after successful add to cart
 * @return bool
 */
public function addToCart(Mage_Checkout_Model_Cart $cart,
    $delete = false)
{
    $product = $this->getProduct();
    ...
    $buyRequest = $this->getBuyRequest();
    $cart->addProduct($product, $buyRequest);
    ....
}
```

You can clearly see that Magento invokes the standard method `$cart->addProduct()` to work with the configuration.

Notice the special method `getBuyRequest()` that's used to retrieve the item's buyRequest. This is a quick wrapper to check several conditions and conveniently compose the result. We can do the same job on our own; however, `getBuyRequest()` eases our life by handling some boring tasks:

1. The `getBuyRequest()` method does routine work required to extract the buyRequest from an item. It remembers the option name (`info_buyRequest`) under which the buyRequest is stored, and finds it for us.
2. Some lists can store products without configuration. Currently only the Wishlist works with incomplete data, but in the future other, similar lists may be developed. The `getBuyRequest()` method creates an empty configuration object in case the configuration option with the buyRequest doesn't exist.
3. This method returns the buyRequest as a `Varien_Object`, rather than as an array. Later it will be much more convenient to use magic methods, rather than check `isset($buyRequest['option_code'])` every time we need a configuration option.
4. This routine automatically copies the current item quantity to the buyRequest. When the product was added to the list, the shopper entered a quantity and the value was recorded within the buyRequest. But the shopper could change this quantity through the interface later, where the product is not reconfigured but only the quantity can be modified. As far as the buyRequest is concerned, it keeps the original quantity, which is probably no longer valid. The `getBuyRequest()` method refreshes the quantity in the buyRequest to the current value associated with the item.

Developers tend to understand code better than words, so let's examine the method's implementation:

Mage_Wishlist_Model_Item

```
/**
 * Returns formatted buyRequest - object, holding request received
 * from product view page with keys and options for configured
 * product
 *
 * @return Varien_Object
 */
public function getBuyRequest()
{
    $option = $this->getOptionByCode('info_buyRequest');
    $buyRequest = new Varien_Object($option ? unserialize($option
        ->getValue()) : null);
    $buyRequest->setOriginalQty($buyRequest->getQty())
        ->setQty($this->getQty());
    return $buyRequest;
}
```

The buyRequest had already been used before Magento 1.5 was released: Quote, Order Management and Recurring Profiles depended on it. With the Composite Products implementation, the API to work with the buyRequest was polished and its usage was made a common practice, not just a way to cope with rare, special cases.

Lite and Full Configuration Parsing Modes for Catalog Products

The module responsible for parsing incoming POST-requests with product configuration, is... Catalog, of course! If you're not familiar with its algorithm of parsing options, let me tell about it briefly so you'll understand the additional changes in Magento made for the implementation of the Composite Products functionality.

After a shopper submits a form to add a product to the Shopping Cart, the Catalog module must prepare the product – i.e., check the incoming POST-request and parse it into options. Each product has a special entity (created via Dependency Injection (DI - http://en.wikipedia.org/wiki/Dependency_injection) called a *Type Model* or *Type Instance*. The Type Model is responsible for the specific behavior of the product due to its product type – Grouped Product, Configurable Product, Bundle and so on. Because different product types have different visual controls to configure them, it's the Type Model's job to work with the configuration.

Magento 1.4 had only one method in Product Type designed to prepare options – `prepareForCart()`. More methods were added in Magento 1.5 in order to extend the power of the Catalog product model.

Typical "add this product to the cart" code still uses the `prepareForCart()` method and looks like:

```
Mage_Sales_Model_Quote

/**
 * Add product to quote
 *
 * Return error message if product type instance can't prepare
 * product
 *
 * @param   mixed $product
 * @return  Mage_Sales_Model_Quote_Item || string
 */
public function addProduct(Mage_Catalog_Model_Product $product,
    $request=null)
{
    ...
    $cartCandidates = $product->getTypeInstance(true)
        ->prepareForCart($request, $product);

    foreach ($cartCandidates as $candidate) {
        $item = $this->_addCatalogProduct($candidate, $candidate
            ->getCartQty());
        ...
    }
    return $item;
}

/**
 * Adding catalog product object data to quote
 *
 * @param   Mage_Catalog_Model_Product $product
 * @return  Mage_Sales_Model_Quote_Item
 */
protected function _addCatalogProduct(Mage_Catalog_Model_Product
    $product, $qty = 1)
{
    // Stick same configurations
    $item = $this->getItemByProduct($product);
    if (!$item) {
        // Configuration didn't stick
        $item = Mage::getModel('sales/quote_item');
       ...
    }
    ...

    $item->setOptions($product->getCustomOptions())
        ->setProduct($product);

    $this->addItem($item);

    return $item;
}
```

As you can see, a product is prepared for the cart (the request is parsed into custom options), checked to see if it can be appended to another item in the cart, then options are set to the Quote item and it is added to the Shopping Cart.

The method `prepareForCart()` doesn't fulfill all the Composite Products' needs. Why? Because it's specifically the `prepareFor`**`Cart`**`()` method. It was designed to be used for a Cart (Quote) entity only. This method performs a **full** validation of an *add-to-cart* request (it checks that all the required options are set).

Thus we needed an additional method to make a lite validation of a request. This lite validation method would check only the entered options and wouldn't demand that all required options be present. Also it would be designed for any module, not just for the Cart. That's why the interface of the product type model was extended in Magento 1.5 (see Fig. 19) with several new methods and constants, and the older `prepareForCart()` routine was modified to redirect calls to the newer method.

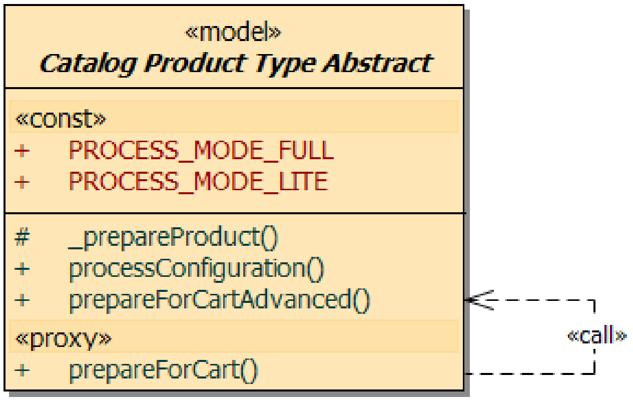


Fig. 19. New and modified entities of Product Type model

The internal implementation of these features is presented in the code below:

```
Mage_Catalog_Model_Product_Type_Abstract

...
/* Full validation - all required options must be set, whole
   configuration must be valid */
const PROCESS_MODE_FULL = 'full';
/* Lite validation - only received options are validated */
const PROCESS_MODE_LITE = 'lite';
...

/**
 * Prepare product and its configuration to be added to some
 * products list.
 * Perform standard preparation process and then prepare options
 * belonging to specific product type.
 *
 * @param  Varien_Object $buyRequest
 * @param  Mage_Catalog_Model_Product $product
 * @param  string $processMode
 * @return array|string
 */
protected function _prepareProduct(Varien_Object $buyRequest,
    $product, $processMode)
{
...
}

/**
 * Process product configuration
 *
 * @param Mage_Catalog_Model_Product $product
 * @param string $processMode
 * @return array|string
 */
public function processConfiguration(Varien_Object $buyRequest,
    $product = null, $processMode = self::PROCESS_MODE_LITE)
{
    $_products = $this->_prepareProduct($buyRequest, $product,
        $processMode);
    $this->processFileQueue();
    return $_products;
}

/**
 * Initialize product(s) for add to cart process.
 * Advanced version of func to prepare product for cart -
 * processMode can be specified there.
 *
 * @param Varien_Object $buyRequest
 * @param Mage_Catalog_Model_Product $product
 * @param null|string $processMode
 * @return array|string
 */
public function prepareForCartAdvanced(Varien_Object $buyRequest,
    $product = null, $processMode = null)
{
    if (!$processMode) {
        $processMode = self::PROCESS_MODE_FULL;
    }
    $_products = $this->_prepareProduct($buyRequest, $product,
        $processMode);
    $this->processFileQueue();
    return $_products;
}

/**
 * Initialize product(s) for add to cart process
 *
 * @param Varien_Object $buyRequest
 * @param Mage_Catalog_Model_Product $product
 * @return array|string
 */
public function prepareForCart(Varien_Object $buyRequest,
    $product = null)
{
    return $this->prepareForCartAdvanced($buyRequest, $product,
        self::PROCESS_MODE_FULL);
}
```

The new `prepareForCartAdvanced()` method was implemented here to let developers control the process of adding a product to the cart. It is the younger brother of the already existing `prepareForCart()` method. The only difference between them is that the advanced method allows control of the validation mode of a product. The simple method, as you see in the source code above, became a dummy in Magento 1.5; it calls `prepareForCartAdvanced()` and passes FULL as the validation mode parameter.

Although the Quote object (which manages the Ordered Items/Shopping Cart list) doesn't usually permit a non-configured product to be added to it, the advanced method is necessary too. It is used on the Create Order page in the Admin Panel to temporarily add a product without configuration to the cart, allowing the store manager complete the configuration later.

The more abstractly named method, `processConfiguration()`, is the twin of `prepareForCartAdvanced()`. Both allow full and lite validations. The reason for making two similar methods is to simplify moving old code to new rails. You can run “Find & Replace” within your module code and change the `prepareForCart()` calls to `prepareForCartAdvanced()`, so semantically your code won't change and all the “add to cart” events will still be clearly visible. However, it is advisable to use `processConfiguration()` for all new development.

File Copy Queue

In the example above, notice the new `processFileQueue()` method:

```
Mage_Catalog_Model_Product_Type_Abstract

...

/**
 * Process product configuration
 *
 * @param Varien_Object $buyRequest
 * @param Mage_Catalog_Model_Product $product
 * @param string $processMode
 * @return array|string
 */
public function processConfiguration(Varien_Object $buyRequest,
    $product = null, $processMode = self::PROCESS_MODE_LITE)
{
    $_products = $this->_prepareProduct($buyRequest, $product,
        $processMode);
    $this->processFileQueue();
    return $_products;
}
...
```

The purpose of this method is to copy all submitted files that are part of the product configuration to their directories inside Magento. **During** configuration processing we don't know whether the whole configuration is valid. That's why we have a separate method to copy these files, executed **after** the whole configuration is **successfully** validated.

In Magento 1.5 every single option adds its uploaded file to the queue array. Options do not copy files anymore. This approach allows you to avoid cases where one option copies a file, the next option fails validation, and the customer's request is rejected – Magento then returns an error to the shopper, but the already copied files are left to forever occupy hard drive space, being completely useless to the system.

If you're developing a configuration option that allows file uploading, the next snippet (taken from the product's file option model) will be handy for you. It shows an option that adds a file to the copy queue in Magento 1.5:

```
Mage_Catalog_Model_Product_Option_Type_File


/**
 * Validate uploaded file
 *
 * @throws Mage_Core_Exception
 * @return Mage_Catalog_Model_Product_Option_Type_File
 */
protected function _validateUploadedFile()
{
    $option = $this->getOption();
    $processingParams = $this->_getProcessingParams();
    ...
    $upload   = new Zend_File_Transfer_Adapter_Http();
    $file = $processingParams->getFilesPrefix() . 'options_'
        . $option->getId() . '_file';
    ...
    $fileInfo = $upload->getFileInfo($file);
    $fileInfo = $fileInfo[$file];

    /**
     * Option Validations
     */
    $upload->addValidator('ImageSize', ...);
    $upload->addValidator('Extension', ...);
    $upload->addValidator('ExcludeExtension', ...);
    $upload->addValidator('FilesSize', ...);

    if ($upload->isUploaded($file) && $upload->isValid($file)) {
        ...
        $fileFullPath = $this->getQuoteTargetDir() . $filePath;

        $upload->addFilter('Rename', array(
            'target' => $fileFullPath,
            'overwrite' => true
        ));

        $this->getProduct()->getTypeInstance(true)
            ->addFileQueue(array(
                'operation' => 'receive_uploaded_file',
                'src_name'  => $file,
                'dst_name'  => $fileFullPath,
                'uploader'  => $upload,
                'option'    => $this,
            ));
    }
    ...
    return $this;
}
```

This option validates several parameters of a file and then passes it to the product's type instance. The type instance has the addFileQueue() method, which receives the *file info* record (associative array) and adds it to the internal queue array. This must contain the following values:

- `operation` – type of operation to perform with file. This can be `receive_uploaded_file` to retrieve an uploaded file, or `move_uploaded_file` to copy a file from an existing path on the disk (this is used to restore previous product configuration).
- `src_name` – path to the source file (file to copy from)
- `dst_name` – path to the destination file (file to copy to)
- `uploader` – instance of `Zend_File_Transfer_Adapter_Http`, which is responsible for receiving the uploaded file (this is ignored for `move_uploaded_file` operations)
- `option` – option instance (can be `NULL`) that will be notified if file copying suddenly fails; this instance must implement the method `setIsValid($isValid)`.

Whenever you make your own product type or add file controls to a product configuration, a good practice is to reproduce the logic as in the code snippet above. During the validation phase your entities should add validated files to the copy queue, not copy them immediately.

Product Configuration Options

Now you know a lot of facts about product configuration:

1. The request is parsed by the Catalog module, creating special instances – options.
2. The options are temporarily stored within the product model as the product's custom options.
3. Custom options affect the product model's behavior (e.g., can influence price calculation).
4. To add a product to an item list, Magento takes the custom options from the prepared product and attaches them to the list item. When the item is saved, it also saves its options to the database.
5. Item options are synced to the item's product custom options.

There is a lot of work with the item's options and the product's custom options. But what are these options? Are they instances of a special class or just values packed into arrays? How are they shared among the Catalog and other modules? What module declares them?

Let's make things clear.

In older versions of Magento, all these options were mere descendants of `Varien_Object`. Modules used the magic method `setValue()` to set their actual validated value. The paired magic method `getValue()` was used to retrieve the value. The options collection was kept in a product as an associative array with option codes as keys and option instances as values.

At a glance, it was hard to understand such amorphous entities. Magento 1.5 has added a lightweight contract, making their usage more strict, clear and reliable:

`Mage_Catalog_Model_Product_Configuration_Item_Option_Interface`

```
/**
 * Retrieve value associated with this option
 *
 * @return mixed
 */
function getValue();
```

Very concise. It shows that no matter how you set a value to an option, whenever you want a product model to work with it, it must implement the `getValue()` method.

Catalog, Shopping Cart, Wishlist and Gift Registry all use item options. These options are quite different in nature. That's why they don't share a common option ancestor, but rather implement the configuration option interface (Fig. 20).

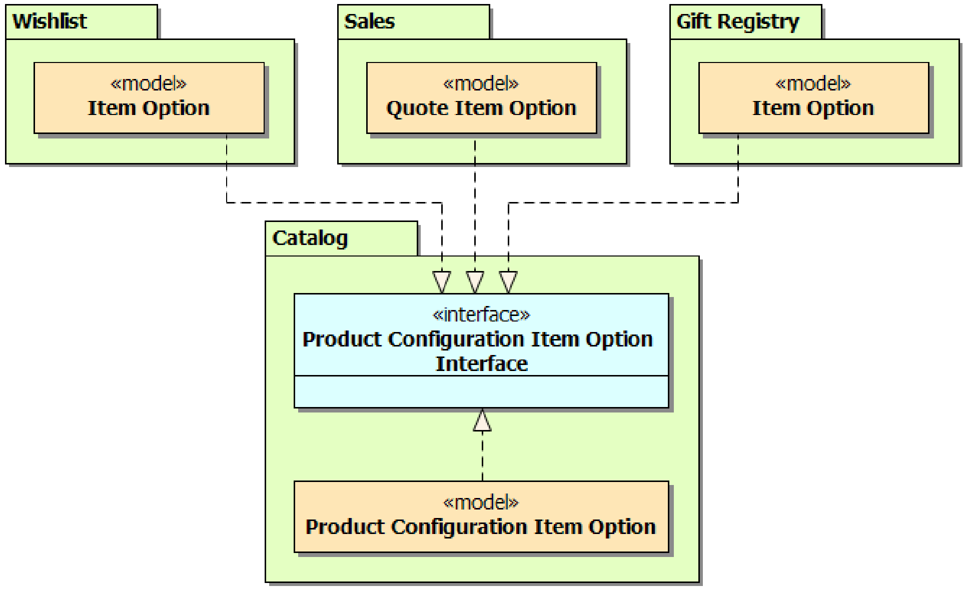


Fig. 20. Configuration options

A few words about diagrams in this guide: I will use human readable shortened names of Magento classes and interfaces. Because a class's full name can be very big, I will change underscores to spaces and use only key words from the class name – the *module name* and several *entity path names*.

The module name will be omitted when it is clear by context. Sometimes the module will be shown as a package. Magento utility words *Model*, *Helper*, *Block*, *Mysql4*, etc. will be omitted too, or replaced by a stereotype. Full class names will only be used in places where it's really necessary for understanding detailed technical information. In that case it will be placed either inside the class block or shown as an additional comment block.

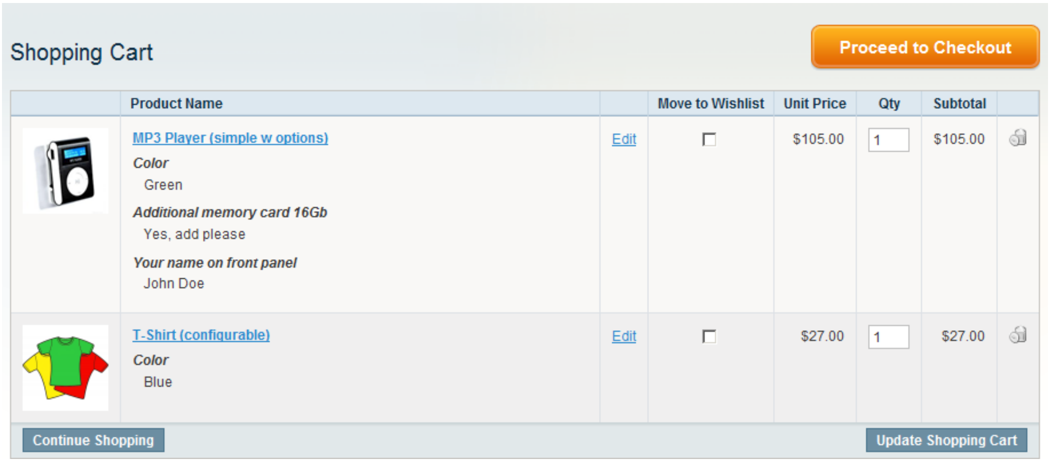
In Fig. 20 you can see an example of such a class name transformation. `Mage_Sales_Model_Quote_Item_Option` became *Quote Item Option*. The namespace *Mage* is omitted, the *Sales* module is shown as a package, and the *Model* is shown as a stereotype.

Currently the item options contract is verified only in one process flow case: when a customer edits a product configuration, and the Catalog model recomposes the product configuration. During this process, the Catalog module needs to work with the current configuration options. The option instances belong to different modules inside the system and are not obliged to have a common ancestor. Thus the Catalog checks the implemented interface to ensure that the option has the required method(s) and can be used in this context. Checking for the contract will be added by the Core team to all future modules in Magento that deal with configuration options.

If you have your own list of items with configured products, you need to create a special class for configuration options and implement `Mage_Catalog_Model_Product_Configuration_Item_Option_Interface`.

Product Configuration Helpers

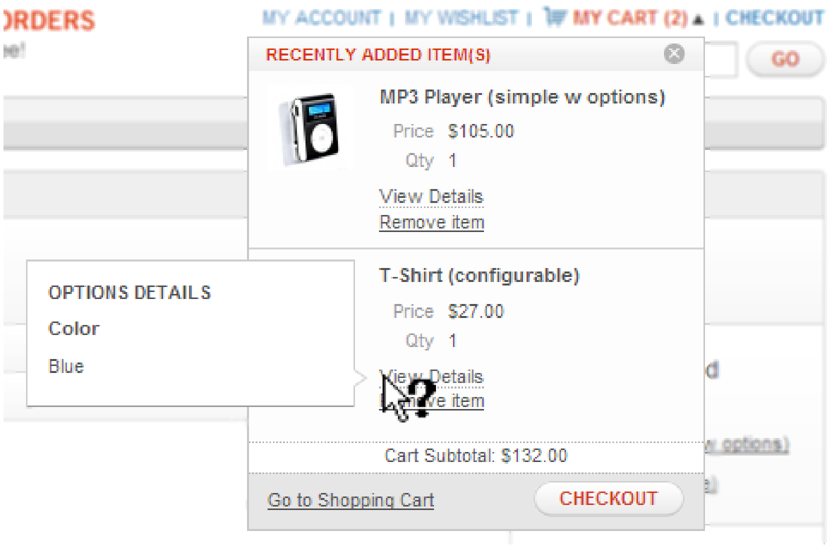
There are pages in Magento where a product is shown to the shopper (or admin) together with its configuration. In older versions of Magento that was only the Shopping Cart block (Fig. 21) and the Shopping Cart floating popup in EE 1.10 (Fig. 22). Their view didn't need to be changed in Magento 1.5.



Shopping Cart

	Product Name		Move to Wishlist	Unit Price	Qty	Subtotal	
	MP3 Player (simple w options) *Color* Green *Additional memory card 16Gb* Yes, add please *Your name on front panel* John Doe	Edit	☐	$105.00	1	$105.00	
	T-Shirt (configurable) *Color* Blue	Edit	☐	$27.00	1	$27.00	

Fig. 21. Product's configuration, as shown at the Shopping Cart



RECENTLY ADDED ITEM(S)

MP3 Player (simple w options)
Price $105.00
Qty 1

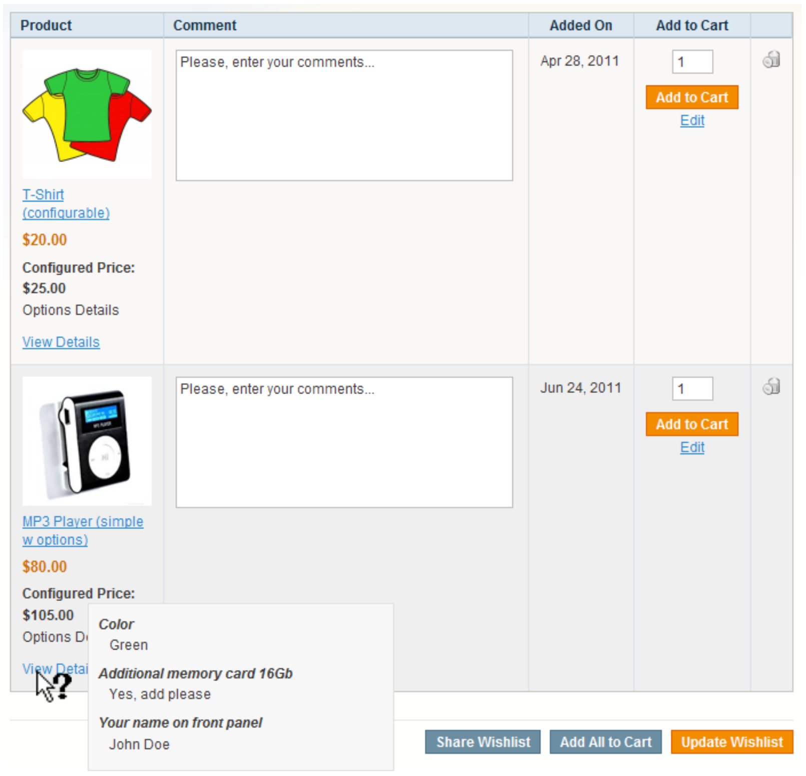
T-Shirt (configurable)
Price $27.00
Qty 1

OPTIONS DETAILS
Color
Blue

Cart Subtotal: $132.00

Fig. 22. Product's configuration, as shown at the Magento EE Shopping Cart popup

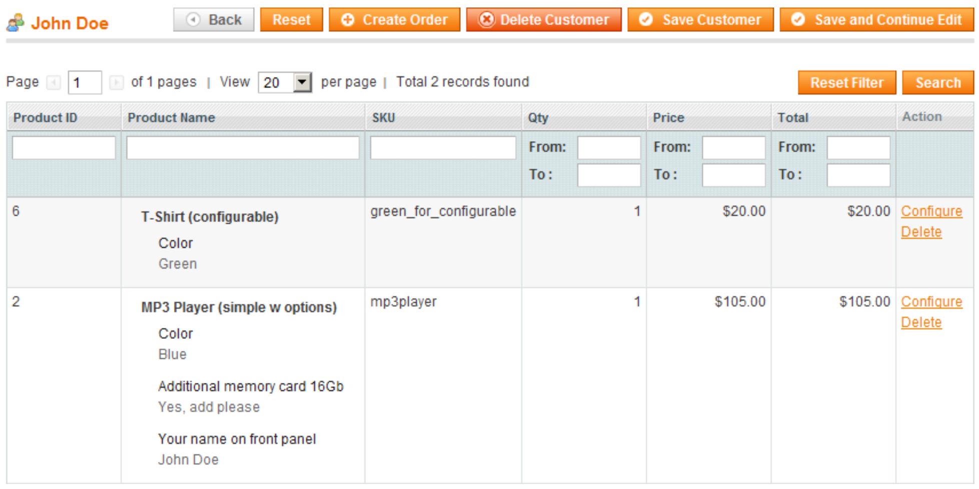
However, in Magento 1.5, more places were added that show a product's configuration. They are: the Wishlist in the frontend (Fig. 23), the Customer's Shopping Cart in the backend (Fig. 24) and the Customer's Wishlist in the backend.



Product	Comment	Added On	Add to Cart
T-Shirt (configurable) $20.00 Configured Price: $25.00 Options Details View Details	Please, enter your comments...	Apr 28, 2011	1 Add to Cart Edit
MP3 Player (simple w options) $80.00 Configured Price: $105.00 Options D… View Detai…	Please, enter your comments...	Jun 24, 2011	1 Add to Cart Edit

Color
Green
Additional memory card 16Gb
Yes, add please
Your name on front panel
John Doe

Fig. 23. Customer's Wishlist at frontend



Page 1 of 1 pages | View 20 per page | Total 2 records found

Product ID	Product Name	SKU	Qty	Price	Total	Action
			From: To:	From: To:	From: To:	
6	T-Shirt (configurable) Color Green	green_for_configurable	1	$20.00	$20.00	Configure Delete
2	MP3 Player (simple w options) Color Blue Additional memory card 16Gb Yes, add please Your name on front panel John Doe	mp3player	1	$105.00	$105.00	Configure Delete

Fig. 24. Customer's Shopping Cart at backend

What do we need to do in order to display product configuration to a user? First of all, we need to dig inside Magento and some specific modules' internals to find all the distinct product option codes. Then, implement the logic of their extraction and form nice display values. But it's not very pleasant to do this in every single place where we want to display configured settings for a product.

I can tell you about my own actual experience. This is how I prepared the example in the earlier chapter (*Item Options)* where we rendered the Configurable Product's configuration via the custom `logCustomerWishlist()` routine:

1. I looked through the code.
2. I found the Configurable Product's type declaration.
3. I searched for the codename of an option code and found `simple_product`.
4. I searched for the option's value manipulation to understand what was stored in that value (it was holding the integer id of a subproduct).
5. Then I needed to implement logic with several IF statements to load the configurable's subproduct.
6. And then I finally got the configuration option's value that's suitable to display to the shopper: the subproduct's title.

If I wanted to display all the options of all 6 product types available in Magento CE (7 in Magento EE), I'd go crazy and this guide would never have appeared on the Web!

To cope with the task of rendering product configuration, Magento 1.5 implemented so-called *Configuration Helpers*. They handle the routine jobs for developers – parse configurations and form arrays of prepared configuration values. These helpers are used in blocks all around the system to output configuration information.

If you develop your own module that implements a product with configuration options, then you definitely need to create your own configuration helper and add it to the blocks in a layout. As a result all your options will be shown in the system. If the product type doesn't feature visually configured options, you don't need to do anything.

If you're building your own theme, you should implement the use of configuration helpers in it. Your v1.4 theme will work with Magento 1.5 without any changes – that's due to carefully preserved backwards compatibility. But, using helpers, you'll get the benefit of automatically rendered options for all new product types added to the system via extensions.

Details of Configuration Helper's Interface

Configuration Helpers were created for lazy developers. Because all of us like to work less and play more Warcraft – these helpers are for us :)

There are 3 built-in Configuration Helpers in Magento CE (4 in EE) – one for each module that adds product types:

- `Mage_Catalog_Helper_Product_Configuration`: Processes Simple, Virtual, Grouped and Configurable Products. Also serves as the default configuration helper.
- `Mage_Bundle_Helper_Catalog_Product_Configuration`: Processes Bundle Products.

- `Mage_Downloadable_Helper_Catalog_Product_Configuration`: Processes Downloadable Products.

> If you develop your own product type, use the Bundle and Downloadable modules as templates for development. These modules do the same task – they act as separate modules, implementing a new product type.

Notice that `Mage_Catalog_Helper_Product_Configuration` is used as a default helper. Whenever a block wants to render item options, and it doesn't find a suitable helper, `Mage_Catalog_Helper_Product_Configuration` is used to extract at least basic information about the product configuration. That's why it's not necessary to develop a configuration helper for your own product type, although it's highly recommended.

All the configuration helpers are tied by contract:

```
Mage_Catalog_Helper_Product_Configuration_Interface

/**
 * Retrieves product options list
 *
 * @param Mage_Catalog_Model_Product_Configuration_Item_Interface
 *     $item
 * @return array
 */
public function getOptions(Mage_Catalog_Model_Product_Configuration_
Item_Interface $item);
```

Visual blocks expect only one routine from a helper – `getOptions()`. This method accepts an item with a configured product and returns an array of options, ready to be displayed to the user.

The `Mage_Catalog_Helper_Product_Configuration` helper should be used by all custom configuration helpers. It is the basic helper, which extracts information about a product's custom options. All other configuration helpers take the result of this basic helper and combine it with configured settings specific to their own product type.

The Downloadable's configuration helper represents an example of this concept:

```
Mage_Downloadable_Helper_Catalog_Product_Configuration

/**
 * Retrieves product options
 *
 * @param Mage_Catalog_Model_Product_Configuration_Item_Interface
 *    $item
 * @return array
 */
```

```
public function getOptions(Mage_Catalog_Model_Product_Configuration_
Item_Interface $item)
{
    $options = Mage::helper('catalog/product_configuration')
        ->getOptions($item); // Basic settings

    // Combine basic settings with Downloadable's specific settings
    $links = $this->getLinks($item);
    if ($links) {
        $linksOption = array(
            'label' => $this->getLinksTitle($item->getProduct()),
            'value' => array()
        );
        foreach ($links as $link) {
            $linksOption['value'][] = $link->getTitle();
        }
        $options[] = $linksOption;
    }

    return $options;
}
```

The helper first receives general information about the configured values of a product, then it goes through the configuration values used by the Downloadable module and adds them to the resulting array. Thus the helper returns a combined array of common configuration settings together with the product type's specific settings. Using the appropriate helper for a product, we can easily get an array of configured values and their human-readable titles.

Configuration Items Processed by Helpers

A few words about `Mage_Catalog_Model_Product_Configuration_Item_Interface`, which must be implemented by items passed to configuration helpers. Because we have different lists that work with configured products (Wishlist, Shopping Cart, EE Gift Registry) and we want to give items to different blocks (e.g., in order to render the Wishlist contents) or configuration helpers (e.g., in order to use the default helper to render any item), we need some clear information on the data this item can provide. That's why Shopping Cart items, Wishlist items and other configured items implement the Configuration Item interface, as shown in Fig. 25.

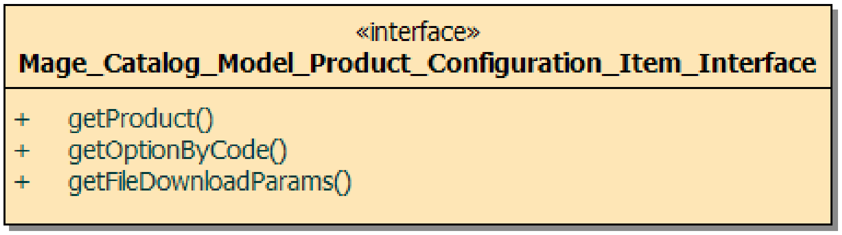

«interface»
Mage_Catalog_Model_Product_Configuration_Item_Interface
+ getProduct()
+ getOptionByCode()
+ getFileDownloadParams()

Fig. 25. Configuration Item interface

Here is a snippet from actual PHP code, where this interface is declared:

```
Mage_Catalog_Model_Product_Configuration_Item_Interface

/**
 * Retrieve associated product
 *
 * @return Mage_Catalog_Model_Product
 */
function getProduct();

/**
 * Get item option by code
 *
 * @param   string $code
 * @return  Mage_Catalog_Model_Product_Configuration_Item_Option_
Interface
 */
public function getOptionByCode($code);

/**
 * Returns special download params (if needed) for custom option
 * with type = 'file'
 * Return null, if not special params needed
 * Or return Varien_Object with any of the following indexes:
 *  - 'url' - url of controller to give the file
 *  - 'urlParams' - additional parameters for url (custom option id,
 *    or item id, for example)
 *
 * @return null|Varien_Object
 */
public function getFileDownloadParams();
```

The first two methods of the interface don't require explanation – they return an item's product and configured options. The third – `getFileDownloadParams()` – returns the url to download a product's files attached to a custom File option. The Wishlist, for example, implements its own controller's action to retrieve files attached to a configuration.

The interface is a requirement on routines to be implemented by any item which has a product with configuration information. You might want to develop such an item class – a new kind of Wishlist, an items archive or something else. In this case your items need to implement the Configuration Item interface so they will be compatible with configuration helpers, rendering blocks and other entities using configuration options in Magento.

On the other hand, this interface allows an item's user (any object that works with the item) to know what general behavior and explicit methods are available. Whenever you develop your own configuration helper, template, logging routine, etc., checking an incoming item's interface allows you to rely on a set of methods the item definitely provides.

Practical Application

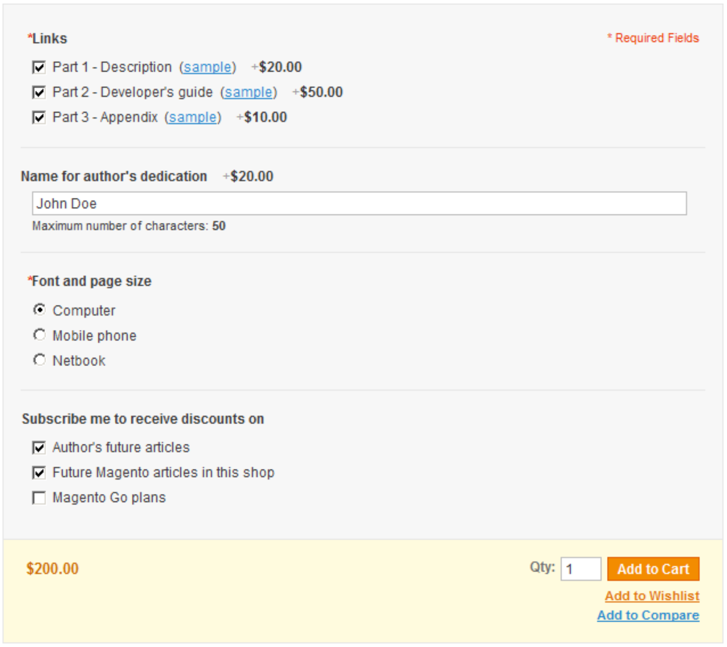
Ok, let's turn from items back to configuration helpers and take a look at how these helpers can be used.

The key feature of the provided configuration helper's interface is the `getOptions()` method. It returns an indexed array of item records, each of which represents an associative array containing `label` and `value` keys. The keys are self-explanatory – `label` stores a string label to display to the shopper, `value` is the option's value or array of values. (There are some other keys the record can contain, but they are outside the scope of this guide.)

Let's examine a specific example of Configuration Helpers usage. Imagine that we are selling a Downloadable Product. It is a PDF containing the mini-book “Product Configuration in Magento”, which has the following characteristics:

- Links are purchased separately.
- One link is called “Part 1 – Description”; this leads to the download of the article's first part.
- The second link is “Part 2 – Developer's guide”.
- The third link is “Part 3 – Appendix”.
- Additionally, the product has a custom option: a text field called “Name for author's dedication”. The shopper can enter his name in this field and we will send him the book printed with the author's inscription “For *[entered name]*...” on the first page.
- There is also another custom option: a radio button titled “Font and page size”. This allows the downloaded PDF to be optimized for viewing on the shopper's particular screen.
- And one more custom option: a checkbox to subscribe and receive discounts for related items.

A shopper adds this product to his Shopping Cart with the options configured as shown in Fig. 26.


*Links — * Required Fields

☑ Part 1 - Description (sample) +$20.00
☑ Part 2 - Developer's guide (sample) +$50.00
☑ Part 3 - Appendix (sample) +$10.00

Name for author's dedication +$20.00
John Doe
Maximum number of characters: **50**

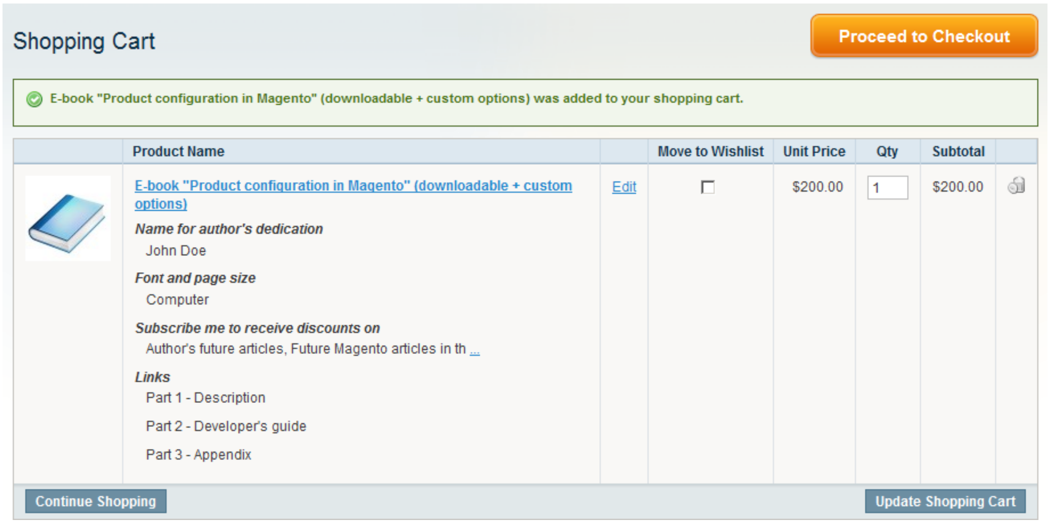
*Font and page size
● Computer
○ Mobile phone
○ Netbook

Subscribe me to receive discounts on
☑ Author's future articles
☑ Future Magento articles in this shop
☐ Magento Go plans

$200.00 Qty: 1

Fig. 26. Configuring the E-book product

Fig. 27 represents a screenshot of the Shopping Cart page showing the product after it was added to the Cart and rendered there.


Shopping Cart

E-book "Product configuration in Magento" (downloadable + custom options) was added to your shopping cart.

	Product Name		Move to Wishlist	Unit Price	Qty	Subtotal	
	E-book "Product configuration in Magento" (downloadable + custom options) *Name for author's dedication* John Doe *Font and page size* Computer *Subscribe me to receive discounts on* Author's future articles, Future Magento articles in th ... *Links* Part 1 - Description Part 2 - Developer's guide Part 3 - Appendix	Edit	☐	$200.00	1	$200.00	

Fig. 27. The E-book product in the customer's Shopping Cart

In Fig. 27 we see that the Downloadable Configuration Helper did its job – it parsed the configuration and returned it to the block responsible for rendering the configuration. The only thing we actually need to program is the routine that renders the array of options and their values inside the block's template.

Let's examine how the work with the configuration helper is done. We'll write a small example and render the Downloadable's configuration options ourselves (in fact, we'll just dump them to output).

```
function dumpQuoteItemOptions($quoteId, $storeId, $itemId)
{
    $quote = Mage::getModel('sales/quote')
        ->setStoreId($storeId)
        ->load($quoteId);
    $item = $quote->getItemById($itemId);
    $helper = Mage::helper(
            'downloadable/catalog_product_configuration'
        );
    $options = $helper->getOptions($item);
    print_r($options);
}
```

The result of `dumpQuoteItemOptions()`, executed for the above-mentioned mini-book configuration, looks like:

```
Array
(
    [0] => Array
        (
            [label] => Name for author's dedication
            [value] => John Doe
            ...
        )

    [1] => Array
        (
            [label] => Font and page size
            [value] => Computer
            ...
        )

    [2] => Array
        (
            [label] => Subscribe me to receive discounts on
            [value] => Author's future articles, Future Magento
articles in this shop
            ...
        )
    [3] => Array
        (
            [label] => Links
            [value] => Array
                (
                    [0] => Part 1 - Description
                    [1] => Part 2 - Developer’s guide
                    [2] => Part 3 - Appendix
                )
        )
)
```

As you can see, it would be a 5-minute task to output these options in an attractive way in your theme. The Configuration helper returns an array of option labels and their values, so the developer's task is just to iterate through them and properly `echo` them in a template.

The only problem with the code above is that it is non-extendable – it can work only with Downloadable products. Of course, we can add several IFs there – “if it's a Downloadable – use Downloadable helper”; “if it's a Bundle – use Bundle helper” and so on. But it will still only provide support to a limited number of product types that we are aware of at the time of development.

How does Magento show options for new custom product types without the need to hack or override core modules?

Integration

The main benefit of Configuration Helpers is that they can be embedded into Magento blocks dynamically. Any extension can add the proper configuration helper during installation, and have custom product type options rendered correctly.

Starting from the first beta, Magento used configurable *Dependency Injection* (DI) for such cases. It's a concept you'll encounter in a lot of places nowadays. For example, Symfony developers get acquainted with it from the first steps in that framework, while building containers.

The idea of DI is that together with your custom module you provide a config describing object classes that will be used dynamically inside the core modules process flow.

The whole Magento layout system is a big example of such DI. All the block definitions that we add in our own layout files provide the DI for page rendering. All the models, controllers and helpers also use DI, due to the ability to rewrite them via config files.

Using the same approach, Magento allows you to build a block and make injections through the layout, by calling the block's methods. This feature is used in Composite Products to create blocks that will load a map of product types and their configuration helpers via layout injection.

To make it clear, look at the way the Wishlist Item options block is implemented:

```
Mage_Wishlist_Block_Customer_Wishlist

/*
 * List of product type configuration to render options list
 */
protected $_optionsCfg = array();

/*
 * Constructor of block - adds default renderer for product
 * configuration
 */
public function _construct()
{
    parent::_construct();
    $this->addOptionsRenderCfg(
        'default',
        'catalog/product_configuration',
        'wishlist/options_list.phtml'
    );
}

...

/*
 * Adds config for rendering product type options
 * If template is null - default will be used
 *
 * @param string $productType
 * @param string $helperName
 * @param null|string $template
 * @return Mage_Wishlist_Block_Customer_Wishlist
 */
public function addOptionsRenderCfg($productType, $helperName,
    $template = null)
{
    $this->_optionsCfg[$productType] = array(
        'helper' => $helperName, 'template' => $template
    );
    return $this;
}

/**
 * Returns html for showing item options
 *
 * @param string $productType
 * @return array|null
 */
```

```
public function getOptionsRenderCfg($productType)
{
    if (isset($this->_optionsCfg[$productType])) {
        return $this->_optionsCfg[$productType];
    } elseif (isset($this->_optionsCfg['default'])) {
        return $this->_optionsCfg['default'];
    } else {
        return null;
    }
}

/**
 * Returns html for showing item options
 *
 * @param Mage_Wishlist_Model_Item $item
 * @return string
 */
public function getDetailsHtml(Mage_Wishlist_Model_Item $item)
{
    $cfg = $this->getOptionsRenderCfg($item->getProduct()
        ->getTypeId());

    $helper = Mage::helper($cfg['helper']);
    if (!($helper instanceof Mage_Catalog_Helper_Product_
Configuration_Interface)) {
        Mage::throwException($this->__("Helper for wishlist options
rendering doesn't implement required interface."));
    }

    $block = $this->getChild('item_options');

    if ($cfg['template']) {
        $template = $cfg['template'];
    } else {
        $cfgDefault = $this->getOptionsRenderCfg('default');
        if (!$cfgDefault) {
            return '';
        }
        $template = $cfgDefault['template'];
    }

    return $block->setTemplate($template)
        ->setOptionList($helper->getOptions($item))
        ->toHtml();
}
```

In the specific example above, the configuration helpers are injected through the call to `addOptionsRenderCfg()`. Whenever it's called with a product type, helper class id and template (optional) parameters, the appropriate configuration helper will be added to the map of product types and their helpers. When the core template renders a configured product's options, it will call `getDetailsHtml()`, choose the correct (or default) configuration helper from that map, and use it to render options for the product type.

Look at the Bundle's layout to see this concept in action. The configuration helper for that product type is injected into the Wishlist rendering process:

/app/design/frontend/base/default/layout/bundle.xml

```
<wishlist_index_index>
    <reference name="customer.wishlist">
        <action method="addOptionsRenderCfg">
            <type>bundle</type>
            <helper>bundle/catalog_product_configuration</helper>
        </action>
    </reference>
</wishlist_index_index>
```

The `<action ...>` instruction tells Magento to call the block's method `addOptionsRenderCfg()` and pass it the parameters residing within the action node. As a result, the product type `bundle` and its configuration helper `bundle/catalog_product_configuration` are added to the map.

The benefits of such an approach are obvious. A block/template pair, once created, can be extended by new modules. Instructions in layout files, deployed together with customizations or extensions, include configuration helpers in the rendering process so there is no need to overwrite templates or blocks in order to add new custom product types to a theme.

You should implement rendering of your product type options in the same way. Create a configuration helper, add it to the blocks that show configuration, and enjoy the nice smooth interface of your store. Magento has no restrictions on method names or parameters used to inject helpers; you're free to develop any methods suitable for your specific blocks and templates.

Summary for Configuration Helpers

Ok, I'm happy that we've made our way through configuration helpers and options rendering. As you can see – there are a lot of words, but not a lot of programming. The concept is simple – add configuration helpers for your product types to existing Magento blocks through layouts and implement configuration rendering in your new blocks the same way – with Dependency Injection. Everything will work nicely, and you will have extensible architecture.

Wishlist Item Configuration

There is nothing complex in the way Wishlist items keep product configuration. If you have worked with Quotes and Quote Items in the Shopping Cart, then you already know everything about the Wishlist.

Implementation is simple (Fig. 28): there is the Wishlist entity, which has Wishlist Items. However, Composite Products added one more entity to this scheme – the Wishlist Item Option. A bunch of these options belongs to a Wishlist Item and keep the configuration for its product. All mentioned entities' data is stored in the `wishlist`, `wishlist_item` and `wishlist_item_options` tables respectively. Two interfaces tie Wishlist entities with the rest of the system, providing reliable inclusion into the products configuration processing.

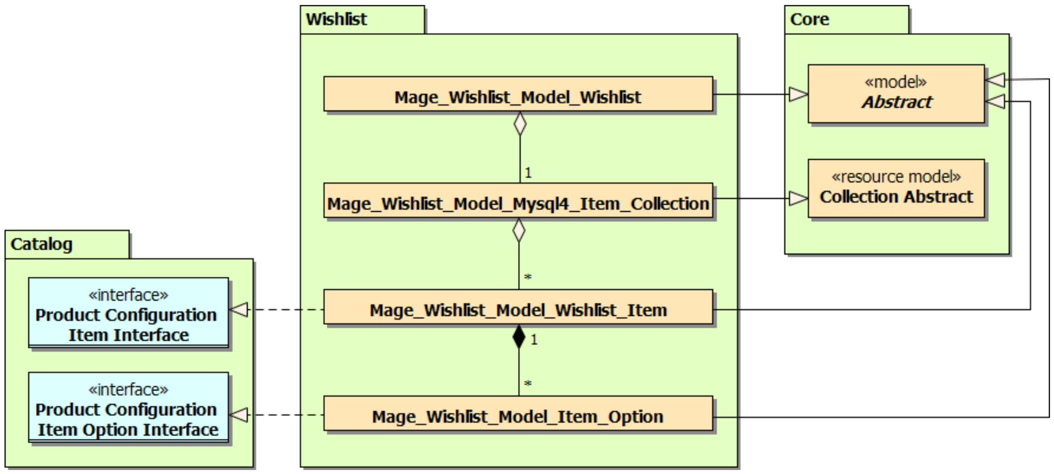

Wishlist
Core
Catalog
Mage_Wishlist_Model_Wishlist
Mage_Wishlist_Model_Mysql4_Item_Collection
Mage_Wishlist_Model_Wishlist_Item
Mage_Wishlist_Model_Item_Option
«model» Abstract
«resource model» Collection Abstract
«interface» Product Configuration Item Interface
«interface» Product Configuration Item Option Interface
1
*
1
*

Fig. 28. Wishlist Item options

Every Wishlist item keeps its options in a protected property and sets them on its product as the product's *custom options*. So the product that you get from a Wishlist item comes with its configuration. It is important to remember this fact, because the item configuration influences the product model behavior (e.g., a Wishlist item's product configured with custom options that cost money will return not the product's original price, but the increased price based on the selected options).

Because Wishlist items have the ability to work with configurations, products can be added to the Wishlist together with their configured values. The `addNewItem()` method is used just for that.

The method has a very typical list of parameters, the same as for similar methods in the Quote or Shopping Cart models. It receives a product (or product id) and buyRequest in the form of POST-data or a `Varien_Object`, then adds that product with its configuration to the Wishlist as a new item.

```
Mage_Wishlist_Model_Wishlist

/**
 * Adds new product to wishlist.
 * Returns new item or string on error.
 *
 * @param int|Mage_Catalog_Model_Product $product
 * @param mixed $buyRequest
 * @param bool $forciblySetQty
 * @return Mage_Wishlist_Model_Item|string
 */
public function addNewItem($product, $buyRequest = null,
$forciblySetQty = false)
```

The `$forciblySetQty` parameter is used for the very specific case when an item is appended to another item and the quantity value in the buyRequest must overwrite the resulting quantity instead of being added to it. Don't worry about this parameter – you'll hardly ever need it.

Another benefit of having a method that adds products with a configuration to the Wishlist is that we can easily integrate the Wishlist and the Shopping Cart. Transferring items between them is now an easy task. Use the appropriate *add item* methods for these item lists, pass them the item's product and the buyRequest, and everything will work smoothly.

Although the Wishlist got many changes during the Composite Products implementation, upgrading Magento CE from v1.4 to v1.5 (or Magento EE from the corresponding v1.9 to v1.10) is not a hard process. The Magento Core team is always concerned about backwards compatibility, so there are not a lot of things that need to be implemented during upgrade and we talk about all of them in this guide. Also, the final chapters provide useful checklists to help Magento module and theme developers update their solutions for the new Magento versions.

Updating Item Configurations

Check out the UML diagram in Fig. 29.

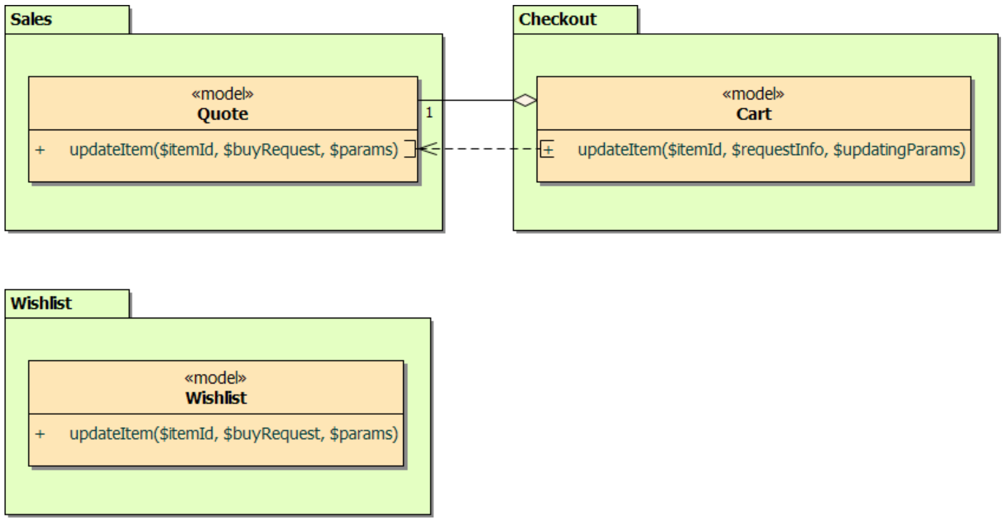

Sales
«model»
Quote
+ updateItem($itemId, $buyRequest, $params)
1
Checkout
«model»
Cart
+ updateItem($itemId, $requestInfo, $updatingParams)
Wishlist
«model»
Wishlist
+ updateItem($itemId, $buyRequest, $params)

Fig. 29. Shopping Cart and Wishlist interfaces for updating product configurations

The available `updateItem()` methods should be used when you need to update a Shopping Cart or Wishlist item. All of them take the same parameters:

- `$itemId` – the id of the item you want to update
- `$buyRequest` or its synonym `$requestInfo` – either a new quantity (simple integer parameter) or a new product configuration to set on the item
- `$params` or its synonym `$updatingParams` – additional parameters for the update operation (they are used by the backend Admin Panel, and are thus described in the Backend section)

A few words about the Checkout Cart model: in the Magento architecture it serves only as a proxy to the Sales Quote model. The Cart is used in the frontend only, its job is to wrap the Quote and improve the shopper's experience during checkout. This model's `updateItem()` method is quite lightweight and is able to automatically tune configurations made by the shopper. It preprocesses the configuration data before passing it to the Quote. The Cart model helps to make the shopper's interaction with the Shopping Cart as pleasant and quick as possible.

Thus it doesn't matter whether we update the Quote item via the Quote model or via the Shopping Cart model – the shopper will see the changes either way. The only difference is that the Shopping Cart is designed to process the shopper's requests in the frontend and provide better interaction there, while the Quote is a universal mechanism independent of the usage area, with strict validation rules.

Summary

The Composite Products implementation mainly concentrates on two things – the product/item options and the buyRequest. The options hold parsed and expanded product configurations, and the buyRequest keeps it in a raw and compacted format. Thus the options are well equipped to work with individual values, while the buyRequest is suited to be transferred between the Shopping Cart, the Wishlist and other item lists.

New internal features, developed for Magento 1.5, include the following: lite product configuration parsing, helpers to visualize it and *update configuration* methods for items and Wishlist item options. These things serve the common purpose to provide a foundation to enable further development and provide a more comfortable interface for the end-user.

Being acquainted with general product configuration management, we are moving on to the two final discussions – actual frontend and backend implementations of Composite Products functionality.

Composite Products in the Frontend

Now we know the internal features developed to make Composite Products work. When talking about specific things that are actually visible to a Magento user, we should divide them into two areas – frontend and backend. The frontend is the place where the **shopper** gets in touch with the system and Magento functionality, and the backend provides the same opportunities to a **store manager** or **admin**.

In this section we will discuss changes made to the frontend view of a Magento store. We will study the View Product / Edit Product page transformation and changes in how the Wishlist handles *view* and *edit* configuration requests.

The View Product Page as the Edit Configuration Page – Process Flow

We have discussed many things already – internal changes for Composite Products, request handling, important interfaces, etc. Now it's time to learn how they are used to transform the View Product page into the Edit Configuration page. Also, we will gain knowledge about the code changes that need to be implemented in custom modules or themes in order to support modifying product configuration.

View -> *Edit* Page Transformation

Do you remember I mentioned that the buyRequest is used for product reconfiguration? This chapter explains that usage.

Previously, before Magento 1.5, the View Product page was rendered by the Catalog Product controller inside the View action. In Magento 1.5 the whole logic of composing the page layout and creating the actual HTML moved to the *Catalog View Product* helper. Therefore it can be used without code duplication by both the View Product and Edit Product Configuration pages. The controller's View action now only prepares parameters for page rendering (checks the product id, sets the current category) and then passes them to the helper, which does the main job of composing the layout and rendering the page.

Let's take the Wishlist as an example of a case where item configurations can be edited (items also can be reconfigured in the Shopping Cart, so everything said here for the Wishlist is also true for the Shopping Cart). Compare the diagram of the *View product* request (Fig. 30) with the diagram of the *Edit item configuration* request, as processed by the Wishlist module (Fig. 31).

You can also compare the actual design of the *View Product* page and the *Edit Wishlist Item Configuration* page in Fig. 32 and Fig. 33.

They are very similar, because they **should be similar,** with only a few visual blocks differing. When modifying the product configuration, a shopper needs to see the same View Product page that he saw when he added the product to his Wishlist (or Shopping Cart). From a usability perspective, it's convenient for the shopper to see the same arrangement of configuration controls (product description, image, prices, etc.). Thus the shopper can easily understand how to reconfigure the product before purchase.

As Magento doesn't like code duplication, it takes the nice approach of reusing the design templates, blocks and layout that were originally created for the View Product page. These design elements are very suitable to render the similar Configuration page.

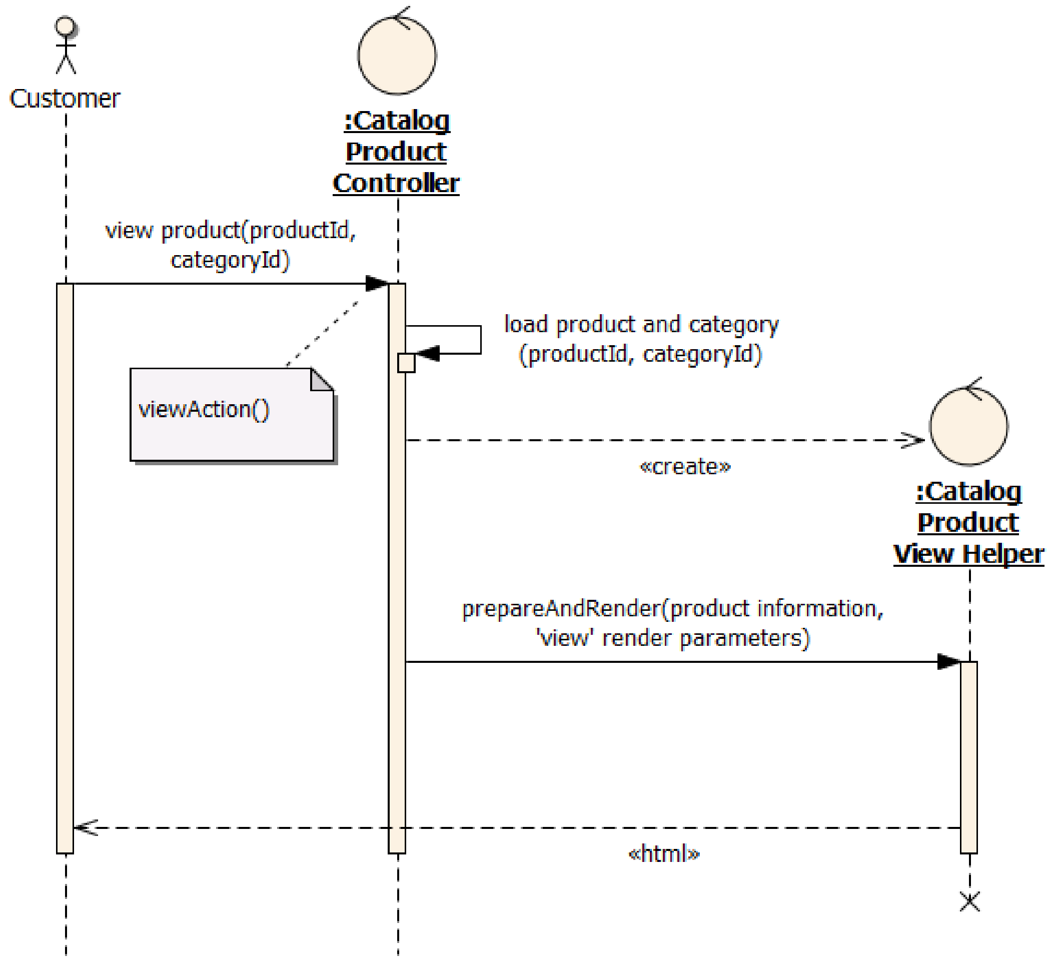

Customer
:Catalog Product Controller
view product(productId, categoryId)
load product and category (productId, categoryId)
viewAction()
«create»
:Catalog Product View Helper
prepareAndRender(product information, 'view' render parameters)
«html»

Fig. 30. View product request being processed

The process flow is as follows:

1. The shopper clicks a product link on the View Category page (or anywhere inside the store).
2. The module controller gets this request inside `viewAction()`.
3. The product and its category are loaded.
4. Product information and current rendering mode are passed to the Catalog View Product helper.
5. The helper renders the page.

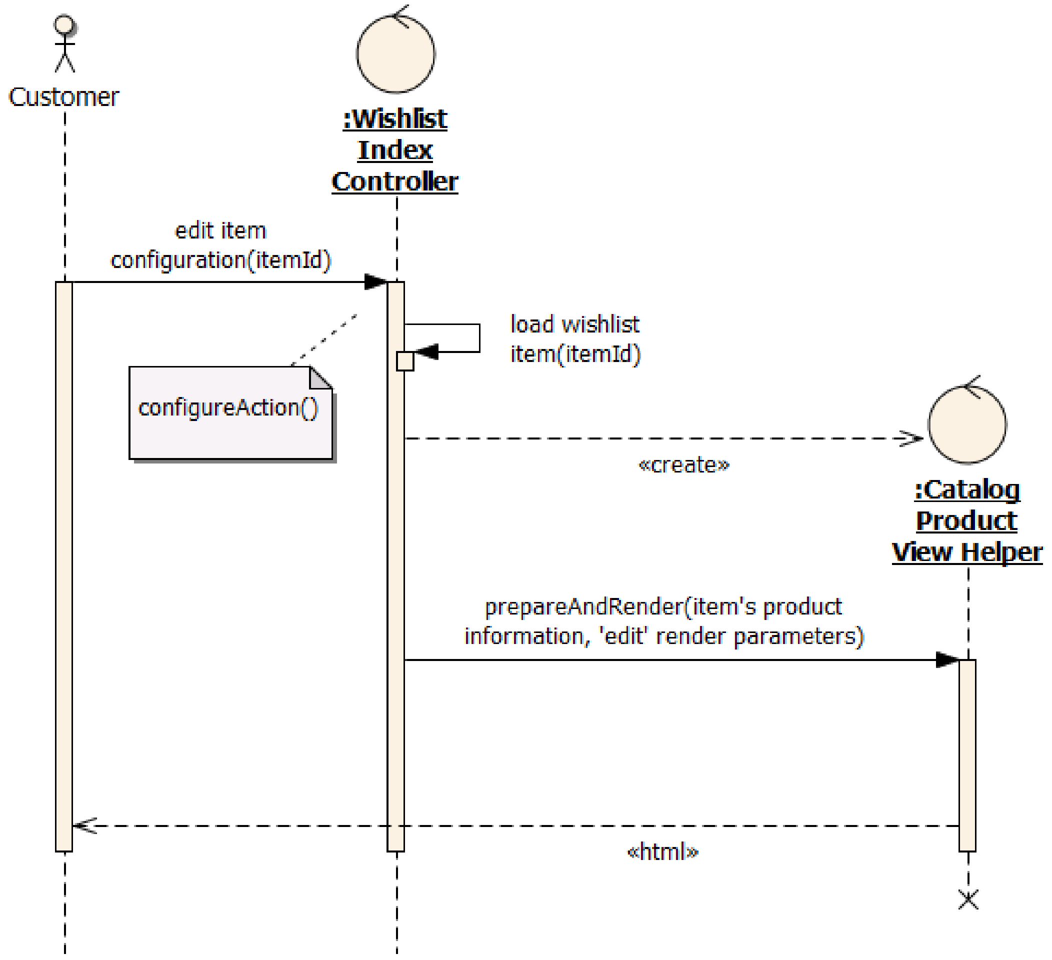
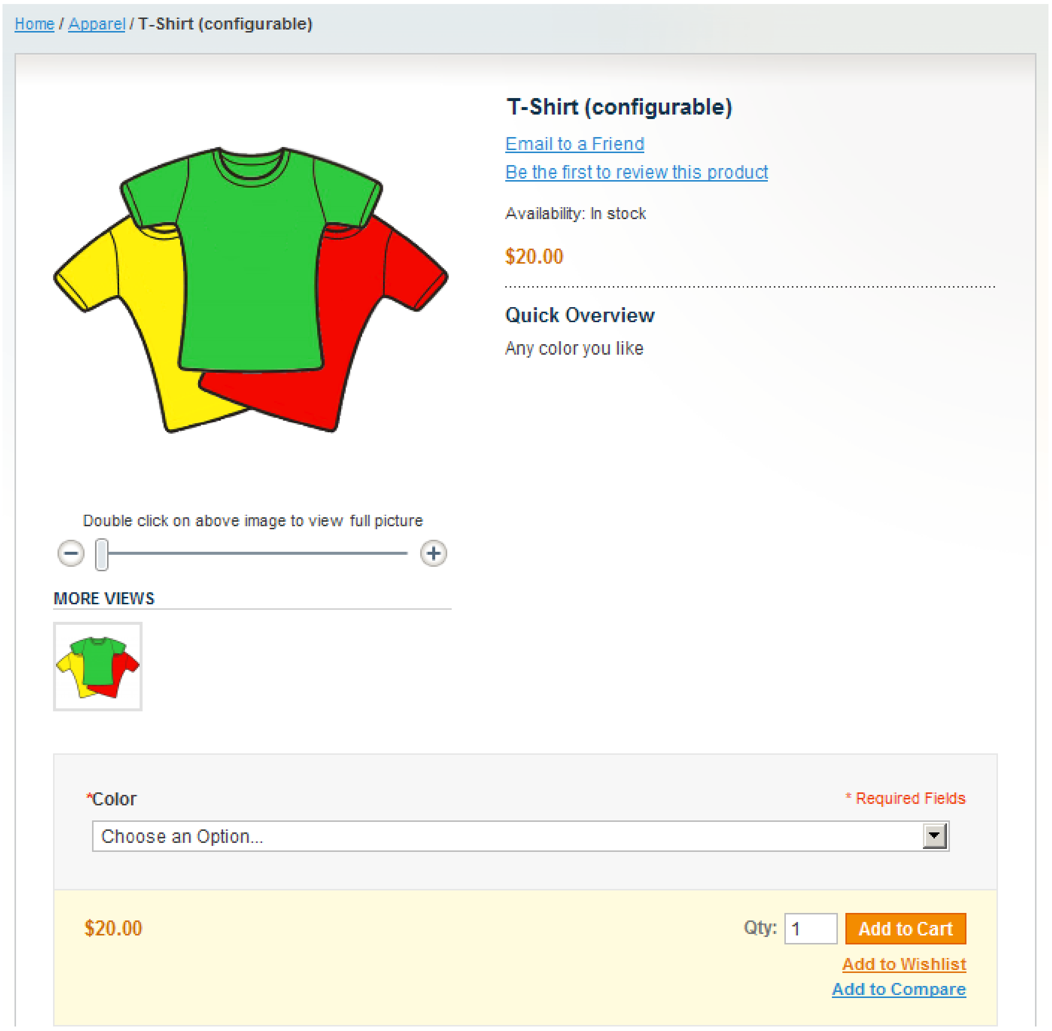

Customer
:Wishlist Index Controller
edit item configuration(itemId)
configureAction()
load wishlist item(itemId)
«create»
:Catalog Product View Helper
prepareAndRender(item's product information, 'edit' render parameters)
«html»

Fig. 31. Edit item configuration request being processed

The process flow is as follows:

1. Shopper clicks the “Edit” link on the Wishlist page.
2. The module controller gets this request inside `configureAction()`.
3. The Wishlist item is loaded and the buyRequest is retrieved.
4. Product information, current rendering mode, and the buyRequest are passed to the Catalog View Product helper.
5. The helper renders the page as follows: the option controls show preconfigured values; controls used to add product to the Wishlist change their titles from “Add To Wishlist” to “Update Wishlist”.



Home / Apparel / T-Shirt (configurable)

T-Shirt (configurable)

Email to a Friend

Be the first to review this product

Availability: In stock

$20.00

Quick Overview

Any color you like

Double click on above image to view full picture

MORE VIEWS

*Color

* Required Fields

Choose an Option...

$20.00

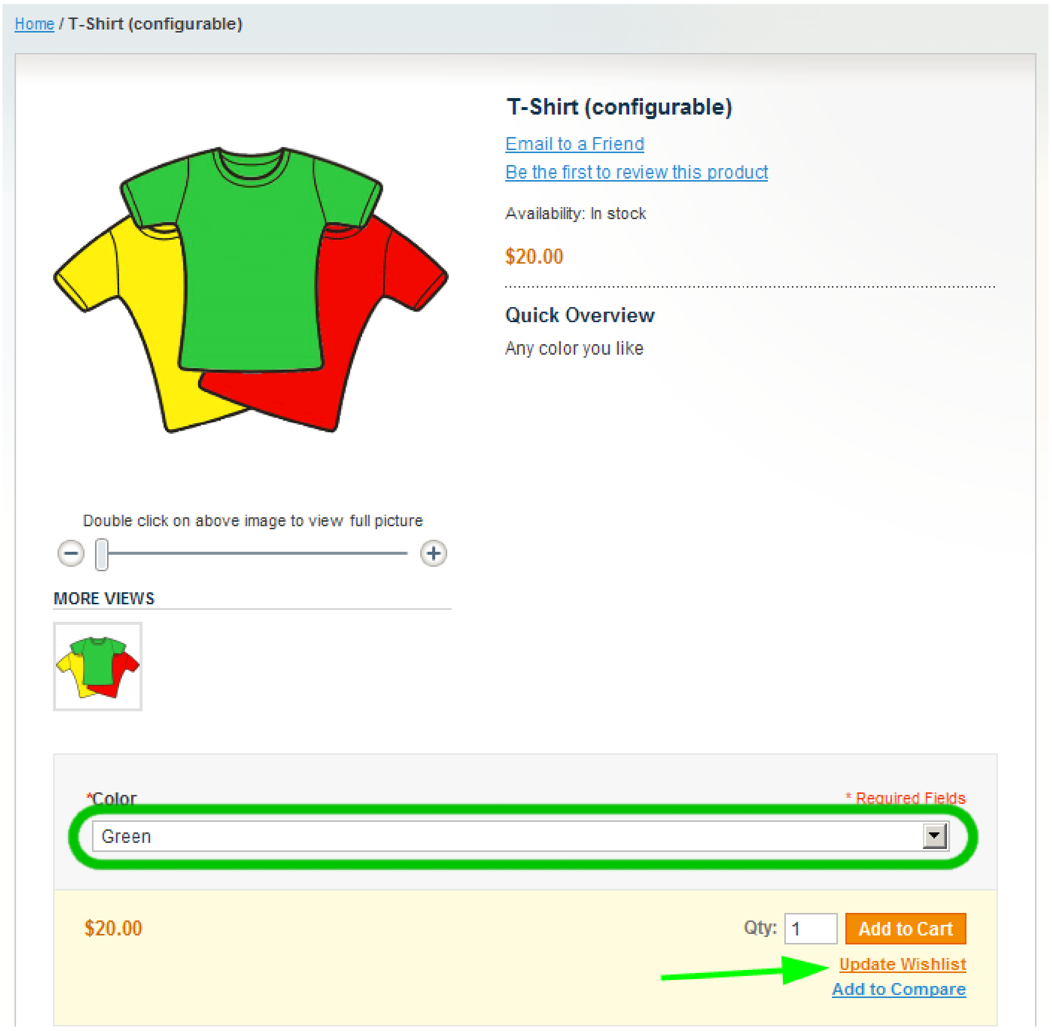
Qty: 1

Add to Cart

Add to Wishlist

Add to Compare

Fig. 32. View Product page



Home / T-Shirt (configurable)

T-Shirt (configurable)

Email to a Friend

Be the first to review this product

Availability: In stock

$20.00

Quick Overview

Any color you like

Double click on above image to view full picture

MORE VIEWS

*Color

* Required Fields

Green

$20.00

Qty: 1

Fig. 33. Edit Wishlist Item Configuration page

Look at the Wishlist layout instructions for the *Edit Wishlist Item Configuration* page:

```
/app/design/frontend/base/default/layout/wishlist.xml

<wishlist_index_configure translate="label">
    <label>Configure Wishlist Item</label>
    <update handle="catalog_product_view"/>
    <reference name="product.info">
        <block type="wishlist/item_configure"
            name="product.info.addto" as="addto"
            template="wishlist/item/configure/addto.phtml"/>
    </reference>
    <reference name="product.info.options.wrapper.bottom">
        <action method="unsetChild">
            <name>product.info.addto</name>
        </action>
        <action method="append">
            <block>product.info.addto</block>
        </action>
    </reference>
</wishlist_index_configure>
```

It shows the ideas I talked about before: the `wishlist_index_configure` layout handle loads a product's View layout and changes some blocks.

You may have the reasonable question, “Why do we need the View Product helper for this scheme? We can manipulate blocks just by using the layout.”

Well, layout isn't a panacea, it only arranges blocks on the page. The helper, in its turn, takes care of the business logic for rendering the View Product page. It prepares the buyRequest to be shown by the option controls, initializes the current category, loads informational messages and so on. The layer provided by the helper is light, but it's really important for the page to be rendered correctly.

Now compare the code that's used by the View Product and the Configure Wishlist Item pages.

View Product page:

```
Mage_Catalog_ProductController

/**
 * Product view action
 */
public function viewAction()
{
    // Get initial data from request
    $categoryId = (int) $this->getRequest()
        ->getParam('category', false);
    $productId  = (int) $this->getRequest()->getParam('id');
    $specifyOptions = $this->getRequest()->getParam('options');

    $viewHelper = Mage::helper('catalog/product_view');
    $params = new Varien_Object();
    $params->setCategoryId($categoryId);
    $params->setSpecifyOptions($specifyOptions);

    // Render page
    try {
        $viewHelper->prepareAndRender($productId, $this, $params);
    } catch (Exception $e) {
        ...
    }
}
```

Configure Wishlist Item page:

```
Mage_Wishlist_IndexController

/**
 * Action to reconfigure wishlist item
 */
public function configureAction()
{
    $id = (int) $this->getRequest()->getParam('id');
    $wishlist = $this->_getWishlist();
    $item = $wishlist->getItem($id);

    try {
        if (!$item) {
            throw new Exception($this->__('Cannot load item'));
        }
        $productId = $item->getProductId();

        $viewHelper = Mage::helper('catalog/product_view');
        $params = new Varien_Object();
        $params->setCategoryId(false);
        $params->setConfigureMode(true);

        $buyRequest = $item->getBuyRequest();
        $params->setBuyRequest($buyRequest);

        $viewHelper->prepareAndRender($productId, $this, $params);
    } catch (...) {
        ...
    }
}
```

The main difference between the View Product and Configure Item pages is that their actions receive different parameters in the http-request. The first one gets the id of a product, the category id and some specific options. The second gets just the list item id. But both these actions use the same `Mage_Catalog_Helper_Product_View::prepareAndRender()` method to show the View Product and Configure Wishlist Item pages (see screenshots in Fig. 32 and Fig. 33.).

Let's take a deeper look into the parameters that `Mage_Catalog_Helper_Product_View::prepareAndRender()` takes, so we will know how to implement Edit Configuration pages for our custom module's product lists in the future. The DocBlock comments briefly describe the parameters:

```
Mage_Catalog_Helper_Product_View

/**
 * Prepares product view page - inits layout and all needed stuff
 *
 * $params can have all values as $params in
 * Mage_Catalog_Helper_Product::initProduct().
 * Plus following keys:
 *   - 'buy_request' - Varien_Object holding buyRequest to configure
 *      product
 *   - 'specify_options' - boolean, whether to show 'Specify
 *     options' message
 *   - 'configure_mode' - boolean, whether we are in Configure mode
 *     to edit product configuration
 *
 * @param int $productId
 * @param Mage_Core_Controller_Front_Action $controller
 * @param null|Varien_Object $params
 *
 * @return Mage_Catalog_Helper_Product_View
 */
public function prepareAndRender($productId, $controller,
    $params = null)
```

The first parameter – `$productId` – is obvious: it's the id of a product we want to render.

The second – `$controller` – is needed to implement a specific interaction between the controller and the layout system. You shouldn't worry about it – just pass your controller instance to this method, and all required internal processes will be performed for you.

The third parameter – `$params` – is much more interesting to us. It's a container (`Varien_Object`) that allows the settings for the page rendering to be specified. These `$params` take part in product initialization, so the method's description also sends us to read the docs for `Mage_Catalog_Helper_Product::initProduct()`. I have combined the keys used by both methods here, in order to provide you with their description:

- `category_id` – the id of the category that will be shown as current on a page. Actually, only the View Product page needs that, because it gives the customer navigation information. If the actual category id is not available, one of the following special values will be used:

  - `NULL` will make the helper guess the category on its own (the current category in Mage *registry* will be checked).
  - `FALSE` will tell the helper that there is no current category. The Shopping Cart and the Wishlist must set the `category_id` to `FALSE` when reconfiguring a product, because there is no related category in this context.

- `configure_mode` – boolean value. Specifies View mode (when `FALSE`) or Configure mode (when `TRUE`) for the page. In Configure mode, the helper will notify the underlying design that the product is rendered in reconfiguration mode, so product option inputs must have initial values pre-set from the buyRequest.
- `buy_request` – the buyRequest of the item being reconfigured (this setting is ignored for View mode). All visual controls render their values based on this setting.
- `specify_options` – boolean value. When set to `TRUE`, it instructs the helper to display the message “Please specify the product's required option(s)” at the top of the page. It's a special parameter used by the Catalog module to inform a shopper that the selected product cannot be added or updated without configuring required options. Generally, as a developer, you won't deal with this setting.

Actually, that's all you need to know in order to successfully implement product reconfiguration. Easy and quick – set some parameters, call the helper method and everything is done. Magento magic in action! :)

The *Catalog View Product* helper declares one public method that I don’t describe in this guide. It is `initProductLayout()`. We don’t need to study this one; it was created only for backwards compatibility and to avoid code copy-paste. The logic inside this method is used not only by the View Product page, but also by other pages handled by the Product controller. So this method is not interesting for us in the context of Composite Products.

Preparing Values for the Edit Configuration Page

One more supplementary routine added in Magento 1.5 to ease working with the buyRequest is the Product's and Product Type's `processBuyRequest()` method. It's used to break the buyRequest into an array of option values so they can be rendered as pre-set values in html controls.

`processBuyRequest()` serves as a validator-and-adapter between the buyRequest and html controls. Because the buyRequest represents the almost raw POST-request, it might contain any data (even unrelated or non-valid data for the product). The main task of `processBuyRequest()` is to check this data and return only valid entries which can be safely used for option control rendering.

In Magento 1.5 this method doesn't do a lot of things – it just extracts values by the keys that product and product type use from the POST-request, and sometimes casts them as integers to prevent security vulnerabilities. In the future this logic may be made more complex, if additional validation is required. The result of this method is a `Varien_Object` with option codes as keys, and option values as values.

Check the diagram in Fig. 34 to see the classes involved in this process.

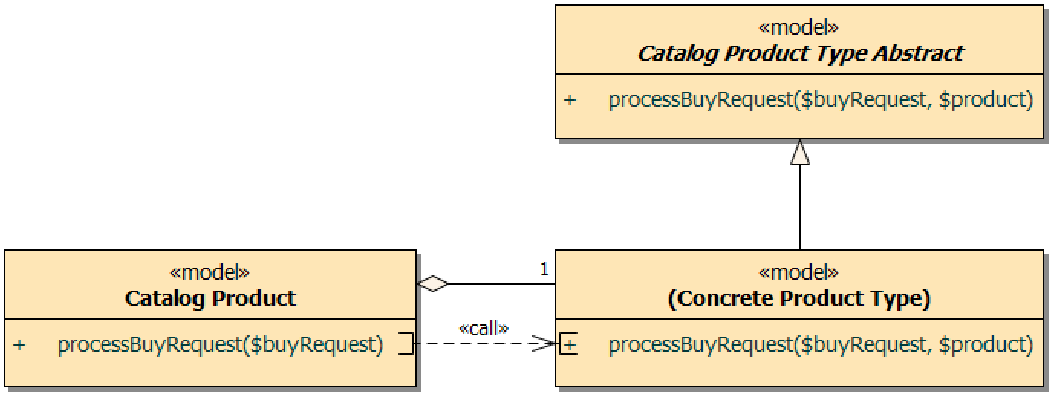

«model»
Catalog Product Type Abstract
+ processBuyRequest($buyRequest, $product)
«model»
Catalog Product
+ processBuyRequest($buyRequest)
1
«call»
«model»
(Concrete Product Type)
+ processBuyRequest($buyRequest, $product)

Fig. 34. processBuyRequest() implementation

And here is a code snippet with this method's implementation in the Catalog Product model:

```
Mage_Catalog_Model_Product

/**
* Parse buyRequest into options values used by product
*
* @param  Varien_Object $buyRequest
* @return Varien_Object
*/
public function processBuyRequest(Varien_Object $buyRequest)
{
    $options = new Varien_Object();

    /* add product custom options data */
    $customOptions = $buyRequest->getOptions();
    if (is_array($customOptions)) {
        $options->setOptions(
            array_diff($buyRequest->getOptions(), array(''))
        );
    }

    /* add product type selected options data */
    $type = $this->getTypeInstance(true);
    $typeSpecificOptions = $type->processBuyRequest($this,
        $buyRequest);
    $options->addData($typeSpecificOptions);
    /* check correctness of product's options */
    $options->setErrors(
        $type->checkProductConfiguration($this, $buyRequest)
    );

    return $options;
}
```

The product model extracts all options that are used by any product of any type – these are the custom options. The product model also asks its type instance to fetch options specific for its concrete product type. Then the product model combines general options and product type-specific options into one buyRequest. As the result we get a nice, clean buyRequest which doesn't contain POST-entries that are not related to product configuration.

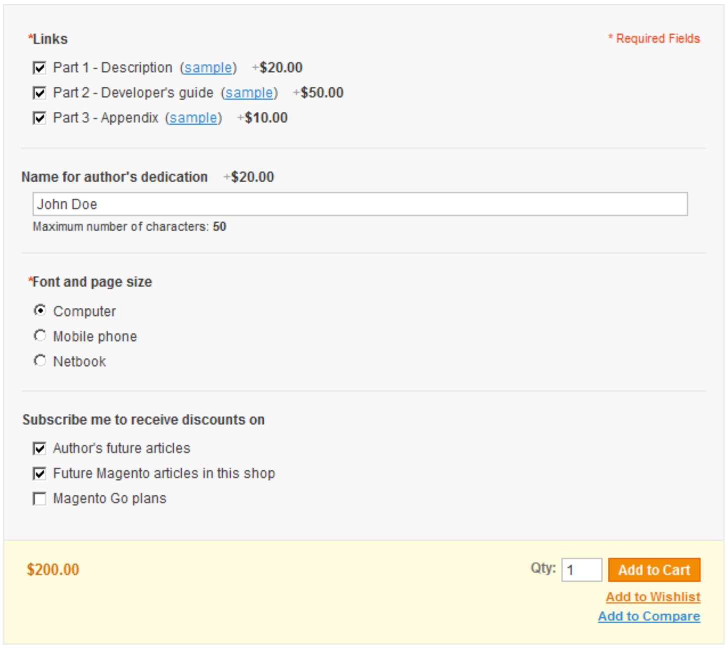
An additional option named `errors` is returned in the result – it holds information about errors that occurred during the validation of the configuration. It can be used to check whether the buyRequest holds incomplete or wrong product configuration.

The `processBuyRequest()` method is very similar across all the product types in Magento. Here is a typical example, taken from the Downloadable product type:

```
Mage_Downloadable_Model_Product_Type

/**
 * Prepare selected options for downloadable product
 *
 * @param  Mage_Catalog_Model_Product $product
 * @param  Varien_Object $buyRequest
 * @return array
 */
public function processBuyRequest($product, $buyRequest)
{
    $links = $buyRequest->getLinks();
    $links = (is_array($links)) ? array_filter($links, 'intval')
        : array();

    $options = array('links' => $links);

    return $options;
}
```

As you can see, Downloadable extracts the specific `links` option from the buyRequest, cleans it and returns it. If the product type uses more options, the algorithm is the same – extract them, clean them and add them to the resulting array.

Do you remember the example of the e-book product we used in the Product Configuration Helpers section? Let's imagine we have configured the product as shown in Fig. 35.



*Links
* Required Fields

[✓] Part 1 - Description (sample) +$20.00
[✓] Part 2 - Developer's guide (sample) +$50.00
[✓] Part 3 - Appendix (sample) +$10.00

Name for author's dedication +$20.00

John Doe

Maximum number of characters: 50

*Font and page size

[✓ Computer]
Mobile phone
Netbook

Subscribe me to receive discounts on

[✓] Author's future articles
[✓] Future Magento articles in this shop
[ ] Magento Go plans

$200.00

Qty: 1

Fig. 35. E-book configuration

After configuring the product, the shopper added it to the Shopping Cart. Out of curiosity, let's check the internal representation of this configuration inside the buyRequest. Here is the code that extracts the raw buyRequest (where `$quoteItem` is the quote item being worked with) and outputs it:

```
$buyRequest = $quoteItem->getBuyRequest();
print_r($buyRequest);
```

And here is the output of this code:

```
Array
(
    [uenc] => aHR0cDovL21hZ2xvY2...,
    [product] => 7
    [related_product] =>
    [links] => Array
        (
            [0] => 1
            [1] => 2
            [2] => 3
        )
    [options] => Array
        (
            [4] => John Doe
            [5] => 5
            [6] => Array
                (
                    [0] => 8
                    [1] => 9
                )

        )

    [qty] => 1
    [original_qty] => 1
)
```

Compare it with the result of passing the buyRequest through the product model's `processBuyRequest()` method:

The slightly modified code:

```
$product = $quoteItem->getProduct();
$buyRequestRaw = $quoteItem->getBuyRequest();
$buyRequestProcessed = $product->processBuyRequest($buyRequest);
print_r($buyRequestProcessed);
```

And its output:

```
Array
(
    [links] => Array
        (
            [0] => 1
            [1] => 2
            [2] => 3
        )
    [options] => Array
        (
            [4] => John Doe
            [5] => 5
            [6] => Array
                (
                    [0] => 8
                    [1] => 9
                )
        )
    [errors] => Array
        (
        )
)
```

We see that after going through the processing routine the buyRequest became much cleaner and more predictable for option rendering controls. Only entries related to the product configuration remain.

Whenever you develop your own product type, don't forget to implement the `processBuyRequest()` method in your product type instance. It must extract the values from the buyRequest and insert them into the resulting array. It's recommended that the keys in this array be the same as those used for html input names, however, for complex controls this isn't always possible. So actually, you're free to use any keys you like. This contract is signed only between your product type model and your product rendering controls; it's their own business to agree on key names.

Theme Modifications

This chapter will be very helpful if you use your own custom theme for Magento (and you surely do) or implement templates for your custom product type. In these cases it's nice to know what kind of modifications are required for the View Product (Edit Configuration) page design in order to adopt the new Composite Products functionality.

The changes that blocks and templates need in order to add Magento 1.5 features to your theme are quick and easy to implement. Even better – some of those features will be enabled in your theme design automatically, without needing any modification. It depends solely on your customization – which module layouts or Magento built-in templates you've overwritten.

Anyway, I have gathered all changes required to support Composite Products on the View Product page as a checklist for you:

1. The “Add to Cart” button changes its title to “Update Cart” when in Edit mode. Find the `<checkout_cart_configure>` handle inside the `checkout.xml` layout and copy it (with its content) to your checkout layout file.
2. The “Add to Wishlist” link changes its title to “Update Wishlist” when in Edit mode. Find the `<wishlist_index_configure>` handle inside the `wishlist.xml` layout and copy it (with its content) to your Wishlist layout.
3. When clicking the “Add to/Update Wishlist” link on the View Product page, the product configuration is submitted by a POST-request to the Wishlist. Find the `template/wishlist/item/configure/addto.phtml` template. The JavaScript there executes when this link is clicked – it changes the form's `action` attribute and submits the form. The options are not validated (remember that the Wishlist can contain incomplete configurations). Implement the same logic in your templates.
4. Option controls show their preconfigured values in Edit mode. If you have overridden

custom option or product type templates, you need to modify them so those controls will use the configured values.

To get more information on the above items, view the corresponding built-in templates and blocks to find out the changes made. A hint: Custom options and Configurable, Grouped, Bundle, Downloadable and Giftcard (Magento EE) product types are affected.

To learn more about rendering configuration values, check the example I have taken from the Text Custom Option's template:

```
template/catalog/product/view/options/type/text.phtml

<input type="text" name="options[<?php echo $_option->getId() ?>]"
   value="<?php echo $this->escapeHtml($this->getDefaultValue()) ?>"
/>
```

and its block:

```
Mage_Catalog_Block_Product_View_Options_Type_Text

...
/**
 * Returns default value to show in text input
 *
 * @return string
 */
public function getDefaultValue()
{
    return $this->getProduct()->getPreconfiguredValues()
        ->getData('options/' . $this->getOption()->getId());
}
...
```

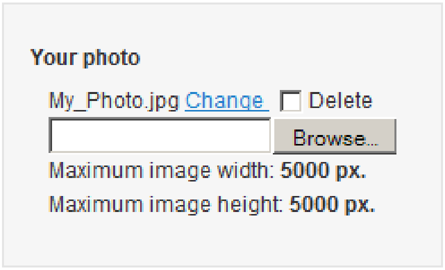
The `getDefaultValue()` method was added to this block in Magento 1.5. The method checks and returns the preconfigured value to be used as the default in the html input.

Actually, similar changes were made to all the above-mentioned templates and blocks that render custom options and product type configuration controls. Now all product option templates use their block's similar methods to get preconfigured values and output them in html inputs.

If you have your own product type that falls under the Composite Products functionality (i.e., it has some configuration controls), then you should add support for the Edit mode in its block/template, so it will display controls with configured options. It's simple – just remember that all configured options are received from the product's `processBuyRequest()` method and are embedded by a controller into the product's preconfigured_values key. So your blocks/templates can grab values from that key by your option path (as set in `processBuyRequest()`) and output these values to your html inputs.

File Inputs in Templates

File input fields, designed to enter product options, are exceptions to the above scheme. We cannot show a default file value in the html and then receive it back. An example of how to cope with this case can be found in the File Custom Option template – `template/catalog/product/view/options/type/file.phtml`. Magento recommends changing the view of a file control in Edit mode, e.g., to show the name of the originally uploaded file and additional links/checkboxes that allow the user to upload a new file or delete the already uploaded file from the configuration (as seen in Fig. 36). If an option is marked as required, the *delete* checkbox should be absent.



Maximum image width: **5000 px.**
Maximum image height: **5000 px.**

Fig. 36. Control of a product file custom option

After the form with the file control is submitted, Magento receives the POST-request with the user's desired action (upload new file / leave everything as is / delete file) and the file input field. The system then changes the configuration according to the checkbox selected:

- If the *new file* action is chosen: the file input field is validated, the old uploaded file is removed and the new one is added to the configuration.
- If the *leave everything as is* action is chosen (no checkbox is selected): Magento does nothing and ignores the file input field.
- If the *delete uploaded file* action is chosen: the previously uploaded file is deleted and the file input field is ignored.

Such a user interface allows the file option to be edited or left as it is. Therefore, Magento doesn't need to set preconfigured data to a file input field (that's just impossible to do).

Wishlist Item Collection

Remember in a previous chapter, where I talked about appended configurations? Let's return to that idea, because it implies one important change everybody should implement in objects that work with Wishlist **product** collections.

The Wishlist **Product** Collection cannot be used anymore

The widely used `Mage_Wishlist_Model_Mysql4_Product_Collection` is deprecated after the release of Magento 1.5. This collection isn't able to serve the Wishlist's needs due to the Wishlist's newly-implemented ability to store **the same product as different items**. The Wishlist **item** collection must be used instead.

Remember the picture with the red and yellow t-shirts, shown at the start of this guide? Let's see it again with some technical information, used to represent internal functionality in Magento.

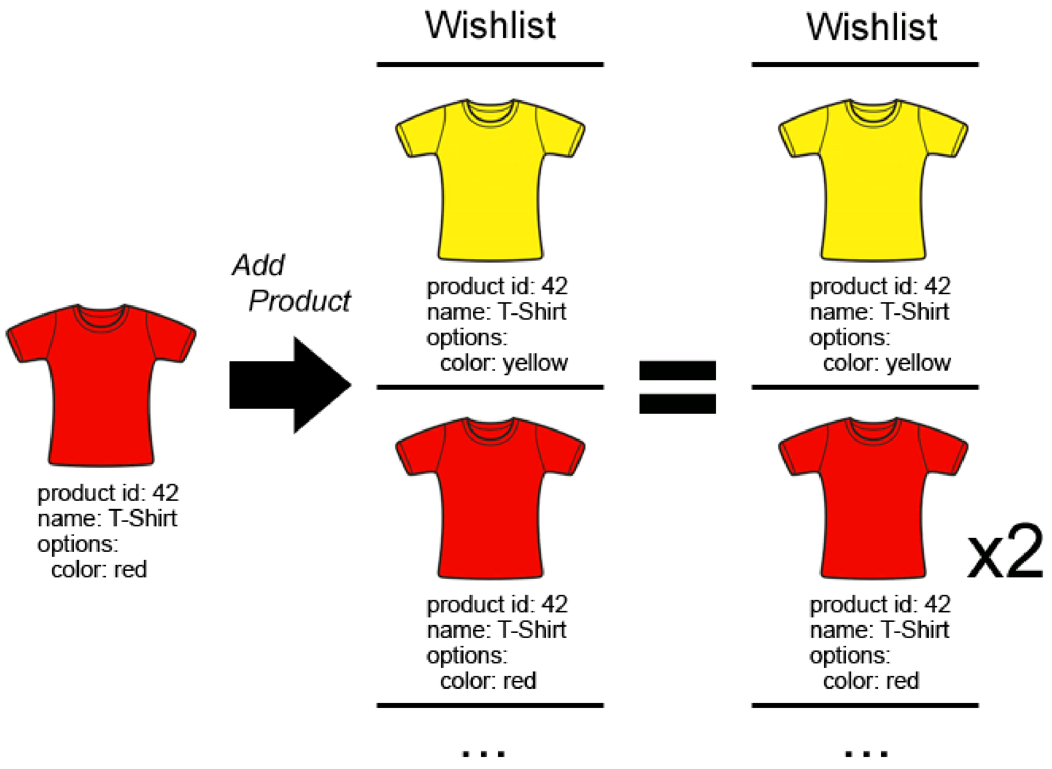


Fig. 37. Appending configurations, technical information included

The t-shirt shown in Fig. 37 is stored twice within the Wishlist items collection. It's the same product – "T-Shirt". It has the same product id (42), but a different product configuration – one t-shirt item has the option 'color' set to 'yellow', the other Wishlist item has the option 'color' set to 'red'. So the product "T-Shirt" is stored under different Wishlist items with different ids. The old product collection isn't able to handle such a case, because it's a **product** collection. As Magento collections do not allow you to have several items with the same id in it, we cannot use the product collection to work with these t-shirts. Generally it means that **product collections don't support the new Wishlist item sets and thus cannot be used with the Wishlist at all.**

Previously the Wishlist wasn't able to store product configurations. Whenever you added a product which was already present in the Wishlist, Magento just discarded the action and did nothing. That's why there was never a case where the same product was present in different Wishlist items. And Magento developers did a really clever thing – they used the Catalog Product collection to work with the Wishlist items. It automatically gave the ability to use all the things previously developed for product collections – from sophisticated filters to applied events.

Of course, there was a separate Wishlist **item** collection, but it was needed so rarely that no one actually used it.

Everything changed in Magento 1.5. The new functionality threw the Wishlist product collection out of business. And if you see errors in Magento like "Item (`Mage_Catalog_Model_Product`) with the same id "N" already exists" or your Wishlist is always empty – then you should definitely fix your theme and custom modules to stop using the Wishlist product collection and start working with the Wishlist item collection.

Switching from the Wishlist product collection to the item collection is easy:

1. Get collection via `Wishlist->getItemCollection()`, not via `Wishlist ->getProductCollection()`.
2. If you use collections by direct instantiation, then change the usage of `Mage_Wishlist_Model_Mysql4_Product_Collection` to `Mage_Wishlist_Model_Mysql4_Item_Collection`.
3. Modify all code that works with items of the changed collection. Because the item collection will be used, its items will be instances of `Wishlist_Model_Item`, not instances of `Catalog_Model_Product`. Basically it means that you should slightly modify your scripts from

```
foreach ($collection as $product) {
        echo $product->getName();
        echo $product->getPrice();
}
```

to

```
foreach ($collection as $item) {
        $product = $item->getProduct();
        echo $product->getName();
        echo $product->getPrice();
}
```

The Magento Core team added several magic proxy methods to the Wishlist collection items – the ones that were most often used on items of the Wishlist product collection. They are `getName()`, `getPrice()` and some others. These methods redirect a call from the Wishlist item to its product. It means that the change described above is not required, and 95% of all custom modules will work with the changed collection even without modifications. However, you should never rely on magic in e-commerce, and it's better to implement the normal usage of the Wishlist item's product.

4. Modify the filters (if any) that you apply to the Wishlist collections, so that they will support items instead of products.
5. Switch your event observers (if any) from Wishlist product collection events to Wishlist item collection events.

Configure and Update Actions in Shopping Cart and Wishlist Controllers

Two newly added actions control the process of modifying product configuration (Fig. 38).

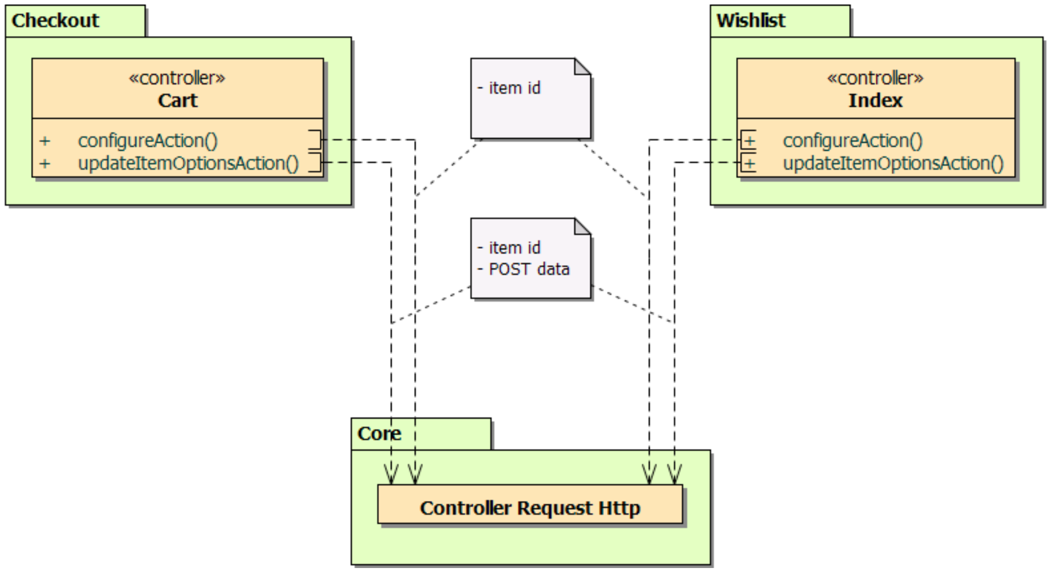


Fig. 38. Controllers' new actions for configuring and updating product configuration

The *configure* action processes the request to start editing the product's configuration. This action has already been described – it loads the product and displays it with pre-set options on the View Product page.

The *updateItemOptions* action is responsible for saving the changed configuration. It receives the POST-request and then calls the list's `updateItem()` method to save the modified item. A good question can arise at this point – why does the controller go through the list's updating routine rather than using the item's `save()` method directly? This is because the item with the new configuration could possibly be appended to another item in the list. In this case the modified item must be deleted, and the appended item's quantity must be increased. Only the list model knows about other items and their configurations, and it must provide solid logic for managing them. That's why the controller updates the item's configuration via a call to the list's `updateItem()` method.

Summary

Let me congratulate you, my reader, for your patience and attention. You've studied the frontend changes and the main implementation details of Composite Products – this knowledge makes you a really powerful developer in the Magento development area.

In order to make you a rockstar, I will provide you with more information on the back-end support for the Composite Products functionality. And at the end of this guide, I will serve a dessert – a theme and product type developer's checklist for working with the Composite Product's functionality and product configurations.

Proceed with me to feel the Power.

Composite Products in the Backend

This part of my story touches the changes made to the Admin Panel. It's a technical part too and will be interesting for Magento developers who work with the system internally, add elements to the pages or modify the logic/actions in the admin backend. It won't be interesting for theme developers who create custom skins. As the Admin Panel rarely uses its own skin, theme developers can just skip reading this part.

There were few pages changed in Magento 1.5 while implementing Composite Products functionality for the Admin Panel. In fact, just two pages were modified in Magento CE, plus one additional page in Magento EE. They are:

- Create Order (CE)
- Customer Information (CE)
- Manage Customer Shopping Cart (EE)

The majority of the changes were made to the Create Order page, and I will focus on it in the upcoming section. The architecture of the AJAX-solution used to manipulate product configurations requires some examples to make it clear. Therefore I describe this solution in the Create Order section, presenting actual examples and code snippets from that page. However, this AJAX functionality is not dependent on the Create Order page – it was designed to be reused in any part of the Admin Panel where product configuration viewing/editing is required.

The Customer Information and Manage Customer Shopping Cart pages are just smaller copies of the Create Order page, so they mostly reuse its code and the AJAX-solution for the manipulation of product configurations. I will dedicate a separate small section to them at the end of this chapter, to highlight the aspects there that differ from the Create Order page.

Ok, I think we're clear enough. Let's proceed to creating orders from the Admin Panel.

Create Order Page

In our world we can divide all people into two major groups – sellers and buyers. People move between these groups and change their roles all the time. At the supermarket you are a buyer, getting milk and bread for breakfast; at your office you are a seller, giving your time and skill in exchange for the monthly salary. The e-commerce website is a replica of an offline shop, with the shopper at the frontend always acting as the buyer, and the admin or store owner at the backend being the seller.

Well, this is true 99% of the time. But there are cases when an admin wants to disguise himself and act on behalf of his customer:

1. A shopper calls the call-center and the operator places an order for him.
2. A shopper isn't able to place an order – he's not comfortable with new technologies.
3. The admin makes an order for a shopper due to a special promotion or to fix a business mistake.
4. Other cases…

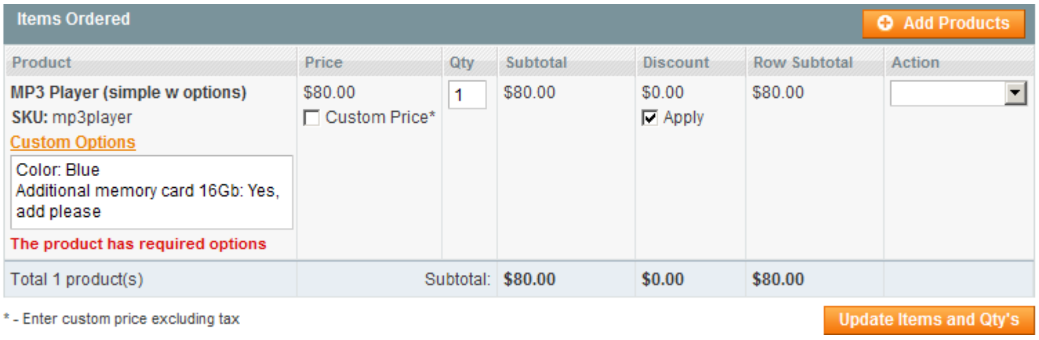
All these things, of course, can be done using the frontend view. The admin goes there, creates a customer account and completes the required tasks. But… there is always a "but" :)

But first of all, the shopper may already have a customer account. Logging into that account would require that the customer tell the admin his password. That is insecure – not acceptable. And second, the admin is a powerful Magento user. He manages the store and needs special abilities to control orders (e.g., the ability to set a custom price for items). So it's much more convenient for the admin to have a special page in the store's backend where he can quickly compose an order for any shopper, while having an extended set of tools to work with prices, items and so on. That's why Magento has a dedicated Create Order page in the Admin Panel.

The Create Order page existed long before Magento version 1.5. It was really useful and well-designed. But there were some things that could be improved to let a manager compose orders faster and better. After Magento 1.0 went public and worked for a long period of time for numerous stores all over the world, we gathered the feedback from store owners and were able to compose a list of "things to improve" . With the feature release 1.5, the Composite Products functionality aimed to remove the drawbacks found in product configuration in the Create Order page. The goal, I must say, was achieved successfully.

Look at the screenshot from Magento 1.4 in Fig. 39. Do you see the text area field under the Custom Options link? Yeah, that was the only way the admin could set options for a product in the Shopping Cart (as of Magento 1.4). He would enter all the options – one per line – with each line consisting of the option label, a semi-colon and the value for that option. Not such a user-friendly interface.

The second drawback was a result of the first one – without a nice interface to set the product configuration, it was impossible to configure more complex product types – Configurable, Bundle, Grouped, Downloadable and Giftcard (EE). Of course, we could provide the same text area option for these product types, but – just for a second – imagine the syntax that might be required for this purpose. Do you think a store owner could enter such data without errors? How many times would the dozens of thousands of admins and store owners all over the world contact Magento developers when trying to find an erroneous comma in a composed configuration? Our devs wanted to sleep well at night, so the solution for Magento 1.0 was neat – all complex product types were removed from the backend order creation. Yes, Magento allowed only the ordering of simple products through the Admin Panel.



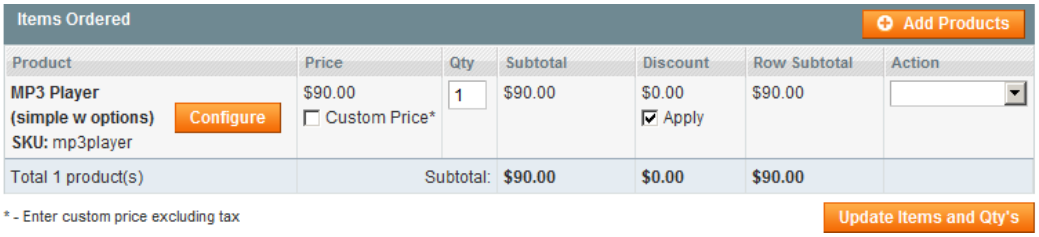
Product	Price	Qty	Subtotal	Discount	Row Subtotal	Action
MP3 Player (simple w options) **SKU:** mp3player Custom Options Color: Blue Additional memory card 16Gb: Yes, add please The product has required options	$80.00 ☐ Custom Price*	1	$80.00	$0.00 ☑ Apply	$80.00	
Total 1 product(s)		Subtotal:	**$80.00**	**$0.00**	**$80.00**	

* - Enter custom price excluding tax

Fig. 39. Entering product's custom options in Magento 1.4

Composite Products in Magento 1.5 fixed that all.

The Create Order page became really convenient for the admin in Magento 1.5. Featuring the same visual configuration controls and rules as used on the frontend, and fast-responding Javascript, it became a pleasant and solid interface to make order creation an easy process.



Product	Price	Qty	Subtotal	Discount	Row Subtotal	Action
MP3 Player (simple w options) **SKU:** mp3player	$90.00 ☐ Custom Price*	1	$90.00	$0.00 ☑ Apply	$90.00	
Total 1 product(s)		Subtotal:	**$90.00**	**$0.00**	**$90.00**	

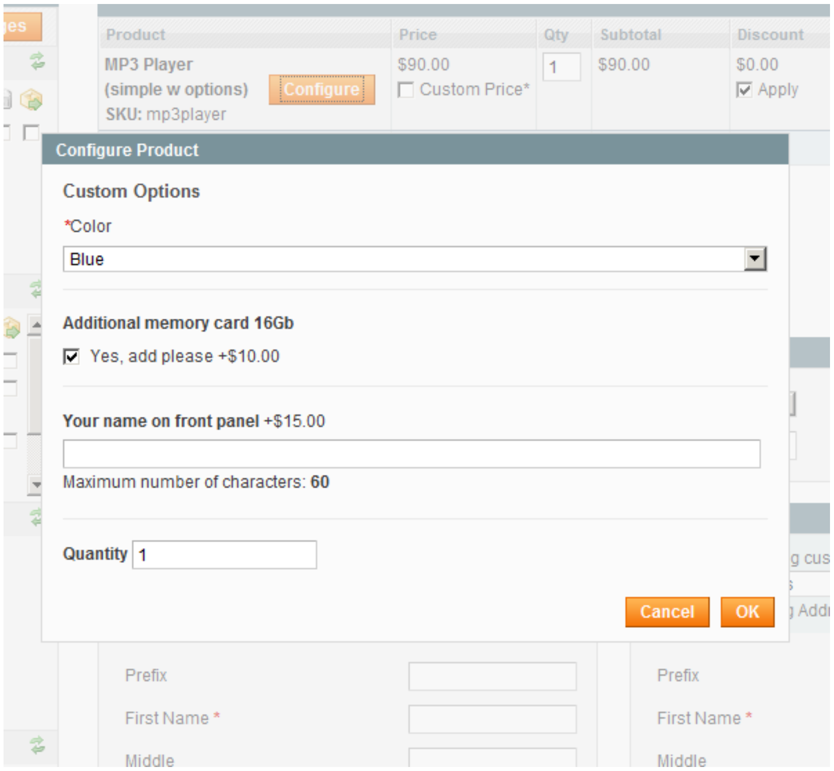
* - Enter custom price excluding tax

Fig. 40. Configure button is available during order creation

Notice the new *Configure* button in Fig. 40. No more text fields here, only convenient user interface elements. The result of clicking this button is shown in Fig. 41.

A nice popup window shows information about product configuration options and provides the same visual controls used in the store's frontend.

Because it's now possible to configure products in a few mouse clicks, Magento allows the admin to manage complex products too. There is no longer the barrier of technical knowledge required to configure them. The nice, new user interface makes work fast and pleasant.



MP3 Player (simple w options) SKU: mp3player — $90.00 — Custom Price* — 1 — $90.00 — $0.00 — Apply

Configure Product

Custom Options

*Color

Blue

Additional memory card 16Gb

☑ Yes, add please +$10.00

Your name on front panel +$15.00

Maximum number of characters: 60

Quantity 1

Fig. 41. Product configuration popup on the Create Order page

Of course, these changes required the Core team's hard work to implement and integrate this functionality. I will explain the internal architecture and concrete programming changes below, so you'll be able to easily provide the same configuration functionality for your own custom products, or extend the Create Product page in the way you need.

Things Developed for the Create Order Page

This is a small chapter. I just want to provide some quick information on the next chapters' topics and help you understand more clearly.

While working with the Create Order page, the following things were created/changed:

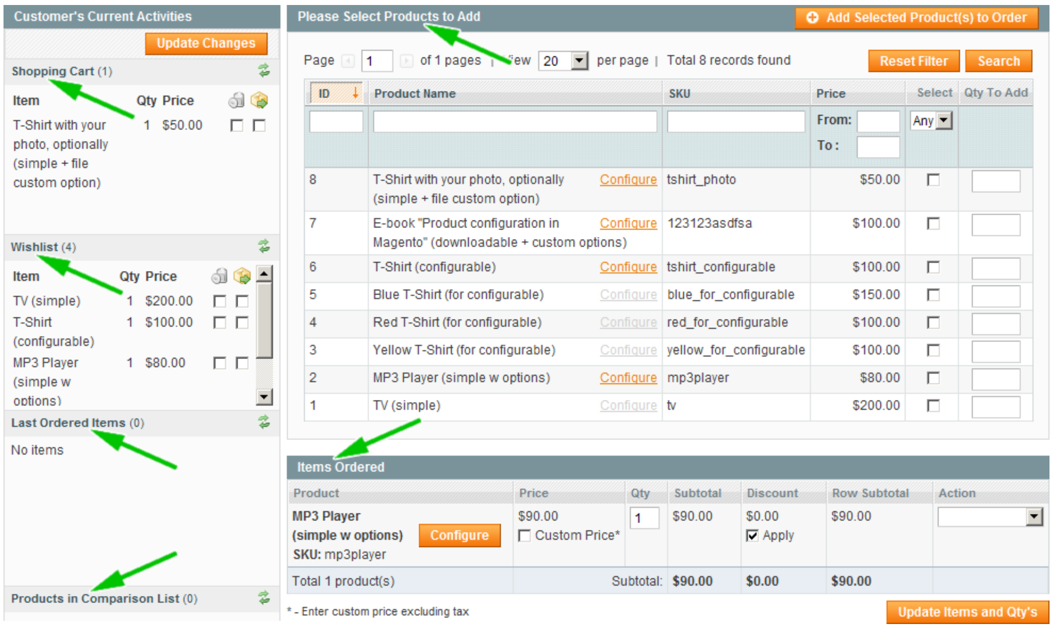
1. *Lists*: This new term was added to the Create Order page in order to describe the workflow for managing sets of products (e.g. Wishlist, Shopping Cart, etc.) and their configurations.
2. Interface: The *Configure* button and links were added to display the configuration popup.
3. Javascript: A popup was added along with an underlying layer to fetch the configuration from the server, show the controls to the store manager, and then submit the modified product configurations back to the server.

4. Visual controls: The frontend controls were used as a starting point, additional changes were incorporated into them to fit specific backend requirements.
5. The server controllers' actions were developed to a) return a unified view with configuration controls and b) save submitted changes.

Lists on the Create Order Page

To start talking about the Create Order page implementation, we need to get acquainted with the new term used there – *List*. Notice that I am using this term only as it relates to the Create Order page. It is a local term, and other Magento modules may use it differently.

A list is a set of items, each having an attached product. The list is managed by a single Magento module. The list knows how to load/edit/delete its items, products and configurations. Lists are represented by visual blocks in the Create Order page (as shown in Fig. 42) which feature different functionality, exposed to manipulate their items.



Customer's Current Activities

Update Changes

Shopping Cart (1)

Item	Qty	Price		
T-Shirt with your photo, optionally (simple + file custom option)	1	$50.00	☐	☐

Wishlist (4)

Item	Qty	Price		
TV (simple)	1	$200.00	☐	☐
T-Shirt (configurable)	1	$100.00	☐	☐
MP3 Player (simple w options)	1	$80.00	☐	☐

Last Ordered Items (0)

No items

Products in Comparison List (0)

Please Select Products to Add

Add Selected Product(s) to Order

Page 1 of 1 pages | ew 20 per page | Total 8 records found

Reset Filter Search

ID	Product Name		SKU	Price	Select	Qty To Add
				From: To:	Any	
8	T-Shirt with your photo, optionally (simple + file custom option)	Configure	tshirt_photo	$50.00	☐	
7	E-book "Product configuration in Magento" (downloadable + custom options)	Configure	123123asdfsa	$100.00	☐	
6	T-Shirt (configurable)	Configure	tshirt_configurable	$100.00	☐	
5	Blue T-Shirt (for configurable)	Configure	blue_for_configurable	$150.00	☐	
4	Red T-Shirt (for configurable)	Configure	red_for_configurable	$100.00	☐	
3	Yellow T-Shirt (for configurable)	Configure	yellow_for_configurable	$100.00	☐	
2	MP3 Player (simple w options)	Configure	mp3player	$80.00	☐	
1	TV (simple)	Configure	tv	$200.00	☐	

Items Ordered

Product	Price	Qty	Subtotal	Discount	Row Subtotal	Action
MP3 Player (simple w options) Configure **SKU:** mp3player	$90.00 ☐ Custom Price*	1	$90.00	$0.00 ☑ Apply	$90.00	
Total 1 product(s)	Subtotal:		**$90.00**	**$0.00**	**$90.00**	

* - Enter custom price excluding tax

Update Items and Qty's

Fig. 42. Different lists on the Create Order page

The following lists are on the Create Order page:

1. *Products*: Contains all catalog products available to be added to an order.
2. *Items Ordered (Quote)*: Contains products already added to an order.
3. *Shopping Cart*, *Wishlist*, *Last Ordered Items*, *Products in Comparison List*, *Recently Compared Products* and *Recently Viewed Products* : These are sidebar lists, showing products the shopper viewed or worked with recently.

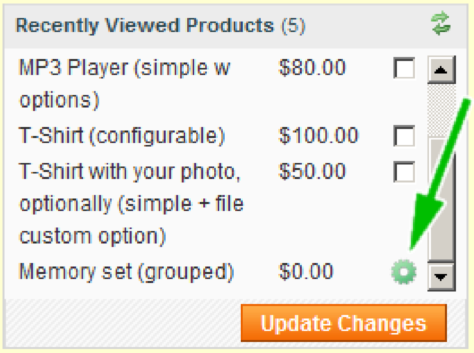
All these lists have different purposes and behavior.

The Products list is a grid with all the products the store owner has in his catalog. The admin can browse through this list, choose products, configure them and add them to the order. All chosen products and their configurations are submitted to a Quote when the admin clicks the *Add Selected Product(s) to Order* button. Until then, the configurations are stored on the client side inside the browser and can be reconfigured at any time.

The Quote list (displayed as the Items Ordered list) is a focal point of this page – it is the Shopping Cart which contains a list of products you're composing to be ordered. The Quote contains the Quote items, which are linked to products and their configurations. Whenever a product is reconfigured in a Quote, its configuration is instantly submitted to the server and the list is refreshed.

The Sidebar lists store information about the shopper's recent activities. The products that the shopper has viewed, ordered or added to the Wishlist or Shopping Cart are shown here. The admin can move a product from any of these lists to the Items Ordered (Quote) list by clicking the Add to Order checkbox to the right of the product, then clicking the *Update Changes* button. The admin does not have the ability to pre-configure products in a Sidebar list – they are added to the order non-configured. Their settings can be entered later, in the Quote item list. Note that items in the Shopping Cart and Wishlist (shown in the sidebar) may contain products which are already configured. Items from these lists added to the Quote will include any existing configurations.

Because Grouped Products are broken into Simple Products when added to the Shopping Cart (this was covered in the “Business Logic Involved” chapter) – it is not possible to add a Grouped Product to a Quote as a non-configured product. That’s why it’s the only product in the sidebar lists which is configured **before** being added to the Shopping Cart. The small gear icon (Configure and Add to Order) beside each Grouped Product (see Fig. 43) replaces the standard Add to Order checkbox and provides access to its configuration popup.



Recently Viewed Products (5)

MP3 Player (simple w options)	$80.00	☐
T-Shirt (configurable)	$100.00	☐
T-Shirt with your photo, optionally (simple + file custom option)	$50.00	☐
Memory set (grouped)	$0.00	⚙

Fig. 43. Gear icon near Grouped Product

While talking about lists on the Create Order page, we have to explicitly state some rules:

- There are several lists on this page, each consisting of items with products attached.

- Item IDs are unique within each list; item IDs are *not* unique within the page (e.g., there can be a Wishlist item with ID = 3, and a Shopping Cart list item that also has ID = 3).
- Product IDs are not unique within a list or within the page.
- Each list belongs to a different module. Therefore, in order to show the configuration popup for the product, the list must have its own controller action which returns the configuration controls in response to the AJAX request.
- Each list can either instantly apply item configurations changed by the admin, or cache them and apply them all later, when the admin clicks a button (such as *Update Changes*) that submits the cached changes.

Architecture on the Client-side

It would be mind-blowing if we tried to think about products, items, lists, configurations, submissions, AJAX, controllers and other things all at once. Let's outline the architecture of the solution, so we can clearly see all involved entities and the connections among them.

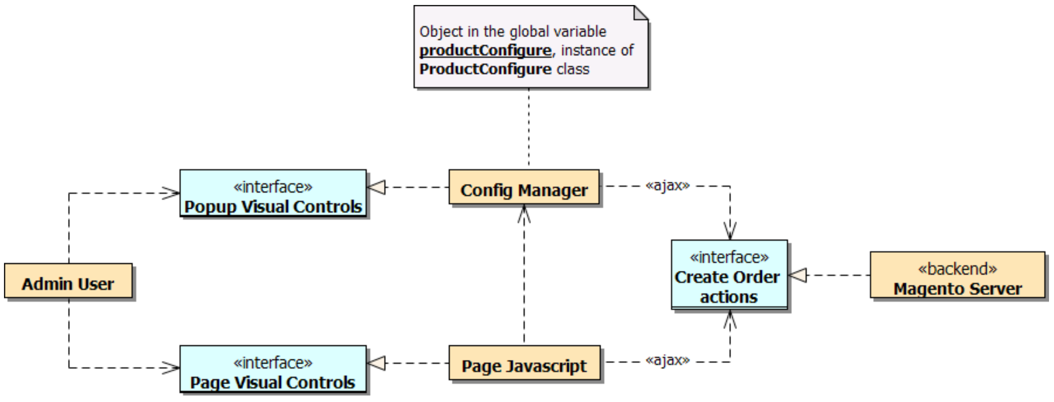

Object in the global variable productConfigure, instance of ProductConfigure class
«interface» Popup Visual Controls
Config Manager
«ajax»
Admin User
«interface» Create Order actions
«backend» Magento Server
«interface» Page Visual Controls
Page Javascript
«ajax»

Fig. 44. Structure of product configuration management environment on the client-side

The principal agent for managing product configurations is what's called the *Config Manager* – a Javascript object stored in the global variable `productConfigure` (an instance of the `ProductConfigure` class). It is the entity with the primary responsibility for popping up the window containing the configuration controls, performing AJAX requests to the backend, and making instant (single item) and postponed (multiple item) configuration submissions.

The admin interacts with the Config Manager either through the page interface (e.g. when he clicks one of the *Configure* links and the page Javscript calls the Config Manager's methods to show item configuration), or directly through visual elements that belong to the Config Manager (e.g., clicking the "OK" button in the product configuration popup).

The Config Manager itself is one of many components in the page's Javascript infrastructure, and it's only responsible for working with product configurations. Thus, as we see in Fig. 44, other Javascript objects communicate with the Magento server without involving the Config Manager. However, all those communications are related to aspects other than product configurations (e.g., refreshing order totals, accepting payment for an order, and so on). We don't discuss those things in this guide, but you should understand that the Config Manager is only a part of the whole page's infrastructure, and it has a special role and dedicated tasks.

Look at the sequence diagram in Fig. 45, which shows a typical scenario in which the admin configures several products, then adds them all to an order:

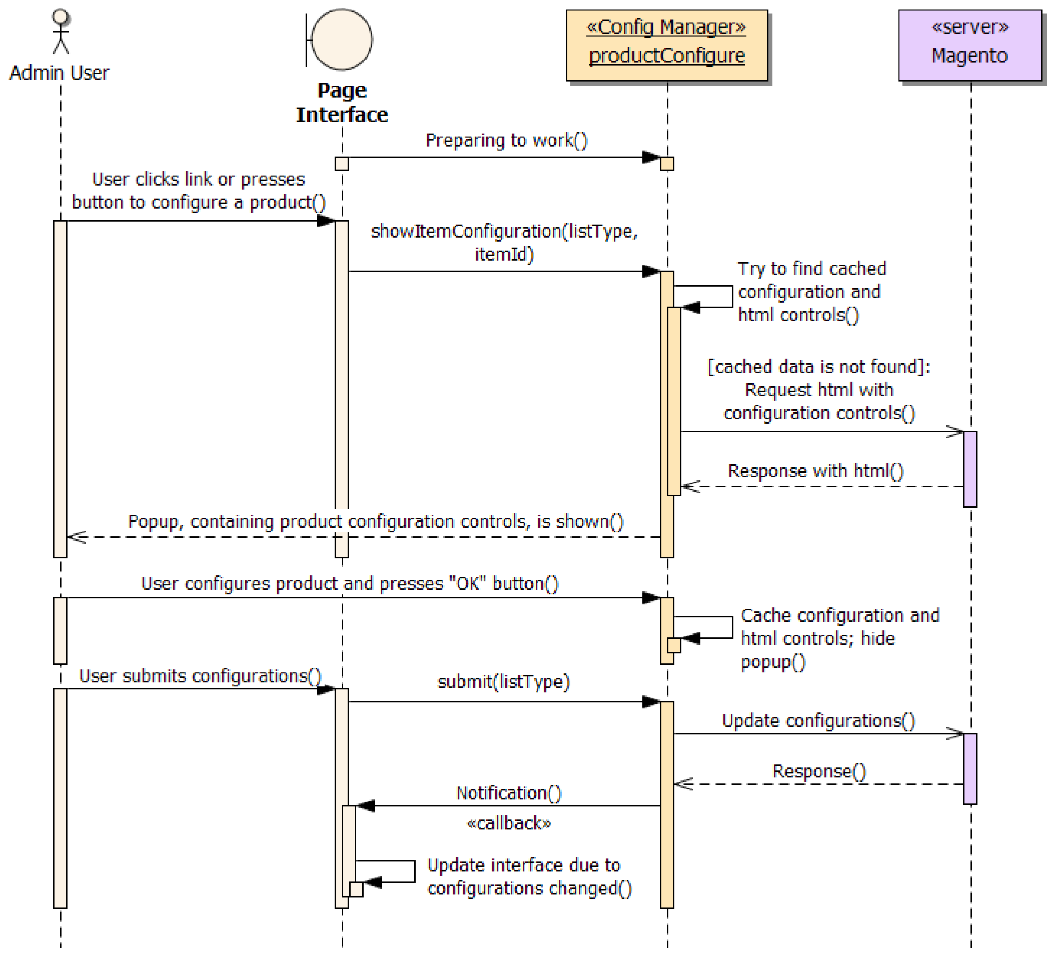

Admin User
Page Interface
«Config Manager» productConfigure
«server» Magento
Preparing to work()
User clicks link or presses button to configure a product()
showItemConfiguration(listType, itemId)
Try to find cached configuration and html controls()
[cached data is not found]: Request html with configuration controls()
Response with html()
Popup, containing product configuration controls, is shown()
User configures product and presses "OK" button()
Cache configuration and html controls; hide popup()
User submits configurations()
submit(listType)
Update configurations()
Response()
Notification()
«callback»
Update interface due to configurations changed()

Fig. 45. Configuring and adding products to an order

As you see in Fig. 45, the Config Manager serves as a layer between the end-user and Magento, issuing/caching requests and managing the client-side popup window with the configuration controls. Caching is used to avoid repeating the same backend requests that retrieve the html with the configuration controls. Once the item html has been received, it should be retrieved from the cache whenever the shopper wishes to modify the same item again.

Every list adds information about itself to the Config Manager – this appears as the “Preparing to work” call in Fig. 45. The passed information includes the list's endpoint at the backend (url) and the required submission type – instant or postponed. *Instant* submissions occur right after clicking the “OK” button in the configuration popup – the form is instantly submitted by the Config Manager to the Magento server, and the list is refreshed to show the changed product configuration. *Postponed* submission is used for lists where it's more convenient for the admin to choose several products, configure them and submit them all to the Shopping Cart in one click (exactly this case is shown in Fig. 45).

The Javascript that serves for the page interface is the only entity which knows about lists, items and controls on the page. Thus it is responsible for issuing the commands “Hey, show this item configuration” or “Submit all my items” to the Config Manager. The Config Manager provides several interface methods for this purpose, and takes further responsibility during the interaction with the user. The flow control is returned via callbacks to the page's Javascript later (after the shopper clicks the *OK* or *Cancel* button in the popup).

The Config Manager's class `ProductConfigure` is defined in the `/js/mage/adminhtml/product/composite/configure.js` file. The Config Manager is instantiated in the same file as the global variable `productConfigure` on the *DOM loaded* event.

Instant and Postponed Submissions

The instant submission shown in Fig. 46 is the simplest case of submission through the Config Manager:

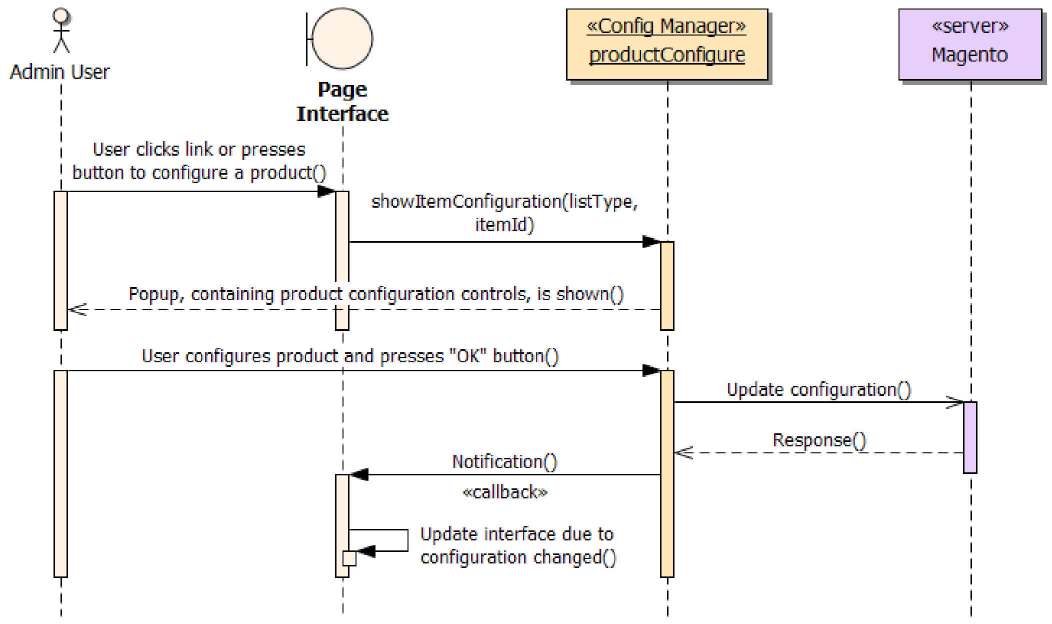

Admin User
Page Interface
«Config Manager» productConfigure
«server» Magento
User clicks link or presses button to configure a product()
showItemConfiguration(listType, itemId)
Popup, containing product configuration controls, is shown()
User configures product and presses "OK" button()
Update configuration()
Response()
Notification()
«callback»
Update interface due to configuration changed()

Fig. 46. Sequence diagram of instant submission

No information is stored on the client-side inside the browser's memory. The single configured item is instantly submitted to the backend. The algorithm is synchronous and straightforward. There is nothing to talk about :)

Instant submission is used for Grouped products, located within sidebar lists on the Create Order page. There the admin clicks the gear icon near the Grouped product's name, configures it, and the resulting product instantly appears in the order grid. This type of submission is also used on the Customer management pages; they will be reviewed later.

Postponed submission is much harder from the developer's point of view (Fig. 47).

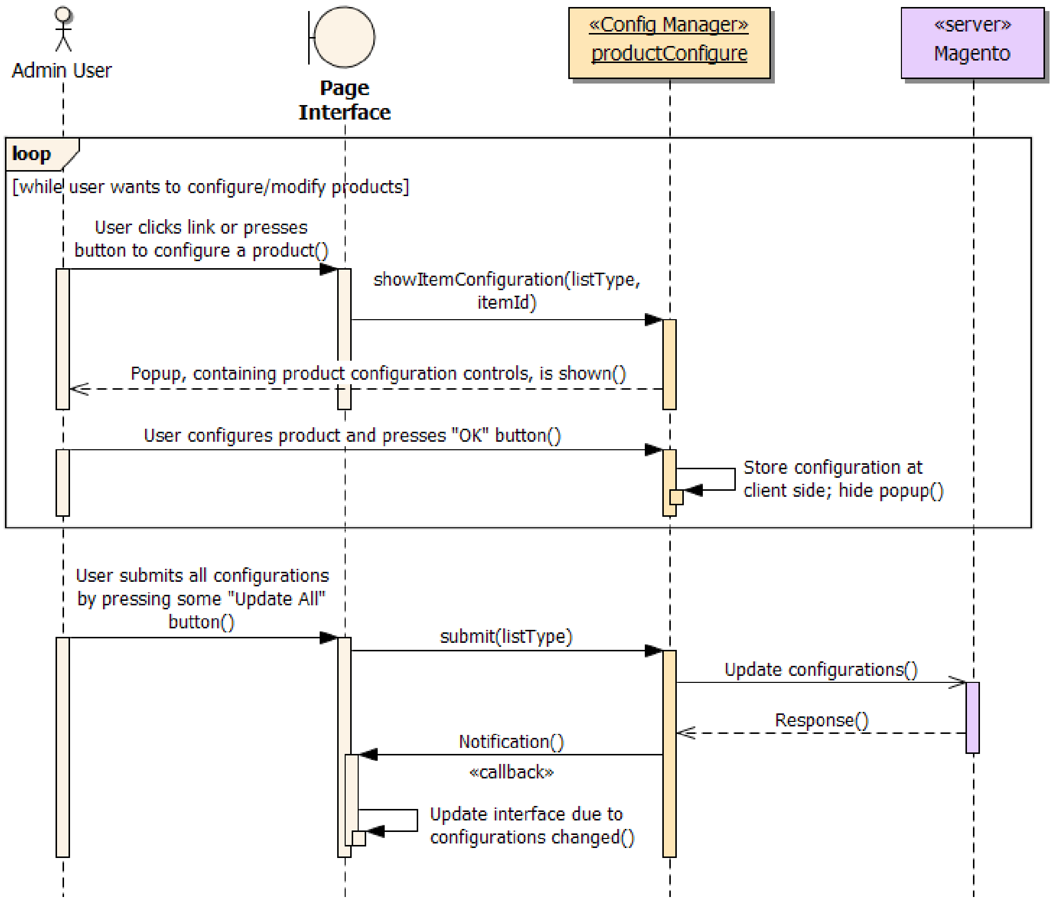

Admin User
Page Interface
«Config Manager» productConfigure
«server» Magento
loop
[while user wants to configure/modify products]
User clicks link or presses button to configure a product()
showItemConfiguration(listType, itemId)
Popup, containing product configuration controls, is shown()
User configures product and presses "OK" button()
Store configuration at client side; hide popup()
User submits all configurations by pressing some "Update All" button()
submit(listType)
Update configurations()
Response()
Notification()
«callback»
Update interface due to configurations changed()

Fig. 47. Sequence diagram of postponed submission

While this action sequence is quite similar to the sequence during the instant submission, it has two important differences:

a. The product configuration is stored inside the browser on the client side until the admin clicks a button (such as *Submit Changes*) to submit the cached changes.

b. The form, being submitted later, includes not just a single product configuration, but a bunch of such configurations.

Why is this so important? Let's see.

First of all, the product configuration is temporarily stored inside the browser. This means that after the store manager clicks the “OK” button in the configuration popup, the Config Manager hides all the controls with the values set up by the admin. Whenever the store manager changes his mind and wants to modify the configuration, the Config Manager restores these controls and shows them within the popup window. The Config Manager also takes care of all the Javascripts that must be executed to support the controls' logic. It's a big piece of work to do.

However, you shouldn't worry about it – the Core team did all that stuff for you. You just need to understand that with postponed submission all the controls for all the products that have been configured are temporarily stored in a special DIV container, hidden inside the page. This technique applies special restrictions on the html+js conjunction used to serve these controls:

1. You cannot freely use the DOM node's *id* field and compose it just from the product identifier. The Javacript, using such a node id, has an excellent chance of finding some other control, because there can be several of the same products (with the same html markup) configured and stored inside the browser, waiting for submission. Whenever your Javascript looks for an element, you should either follow best practices and use unique ids for controls (e.g., `wishlist_item_17_checkbox`), or violate DOM-rules and use duplicating ids, but search for the right element on your own.
2. Timers that interact with the controls should be used with caution. After a user finishes configuring a product and clicks the “OK” button, the controls are hidden and removed from their usual location. They should not be activated, checked or modified by timers. You can use the Config Manager's callbacks (described later in this guide) to cancel timers when the popup is closed.
3. The Javascript coming with the controls is evaluated each time the popup is shown to the admin. This script should not unconditionally set initial values and capture events. When the configuration is **re-shown** on the user's request, the javascript should check the current state of the controls and **leave them as they are**. During your scripts' execution the controls can use the Config Manager's `restorePhase` property to know whether they are being executed for the first time or not.
4. All the Javascripts are executed inside the `eval()` method. Check the PrototypeJS documentation (http://api.prototypejs.org/language/String/prototype/evalScripts/) to learn the specifics of writing code for it.

Here is a code snippet from the Configurable product's template, which avoids the issues described above. The template contains controls intended to be shown inside the AJAX configuration popup. The Javascript is responsible for observing the controls, therefore it starts its own class to capture and react to user events.

```
<script type="text/javascript">
    var config = <?php echo $this->getJsonConfig() ?>;
    if (window.productConfigure) {
        config.containerId = window.productConfigure
            .blockFormFields.id;
        if (window.productConfigure.restorePhase) {
            config.inputsInitialized = true;
        }
    }
    ProductConfigure.spConfig = new Product.Config(config);
</script>
```

We see that the Configurable product type uses the global variable `window`, as suggested by the Prototype documentation, in order to make sure that the work is done with the right scope for variables. Also it checks the Config Manager's `restorePhase` property to notify its scripts that they are not shown the first time, and thus must preserve the values in the controls, and not reset them to the default ones.

The second thing to understand about postponed submission is that it includes many products and configurations. The Config Manager takes care of input names to avoid collisions inside the submitted POST-data, because configuration controls of different items may have the same input names. Digging into the internals, you can see that the Config Manager puts every item's set of controls in a separate cell in the POST-array. Thus a namespace is built for every item, whose controls are renamed from `original_name` to `item[$itemId][original_name]`. Thus the controller at the server backend can work with the `$_POST['item'][$itemId]` entry, which holds the entire submitted configuration for an item.

You can see an example of handling multiple configurations during the *Update Quote items* operation on the Create Order page. This operation is executed every time a user configures several products in a Quote and clicks the "*Update Items and Qty's*" button. The Controller knows that the items' configurations come in the `item` entry of the POST-array, thus passing them to the processing model:

Mage_Adminhtml_Sales_Order_CreateController

```
/**
 * Processing request data
 *
 * @param string $action
 *
 * @return Mage_Adminhtml_Sales_Order_CreateController
 */
protected function _processData($action = null)
{
...
    /**
     * Update quote items
     */
    // Check flag, that quote items are updated
    if ($this->getRequest()->getPost('update_items')) {
        // Items' configurations
        $items = $this->getRequest()->getPost('item', array());
        ...
        $this->_getOrderCreateModel()->updateQuoteItems($items);
    }
    ...
}
```

The processing model knows that multiple items are submitted, iterates through them and updates items according to their POST-data:

```
Mage_Adminhtml_Model_Sales_Order_Create

/**
 * Update order quote items
 *
 * @param  array $data
 * @return  Mage_Adminhtml_Model_Sales_Order_Create
 */
public function updateQuoteItems($data)
{
    // $POST['item'] is received as $data
    if (is_array($data)) {
        foreach ($data as $itemId => $info) {
            $this->getQuote()->updateItem(
                $itemId, new Varien_Object($info)
            );
...
```

File Controls and Embedding Data into the buyRequest

File controls are exceptions to the general control management scheme used by the Config Manager. These controls cannot be cloned easily, and default values cannot be pre-set on them due to browser security restrictions. Otherwise what would you think of a browser, if it were allowed to insert any file from your computer into the submitted form? That's why the Config Manager uses a special technique to work with file controls.

Also, file controls do not fit the common namespace separation scheme used by the Config Manager, because PHP doesn't parse input names into arrays in `$_FILES`. Thus the Config Manager uses plain prefixes for file inputs instead of arrays, and renames file inputs from `original_name` to `item_$itemId_original_name` during postponed submission. Notice that during instant product configuration submission no controls are renamed, as only one product is submitted and collisions are therefore impossible.

When preparing the product and validating the configuration, the file option model should have the information about the file control namespace prefix so it will be able to find the new configuration in the `$_POST` and `$_FILES` arrays. There we come to a conclusion that meta-information should be passed to the file validation models together with the buyRequest. So Magento 1.5 got the ability to embed additional data inside the buyRequest (Fig. 48) exactly for this case.

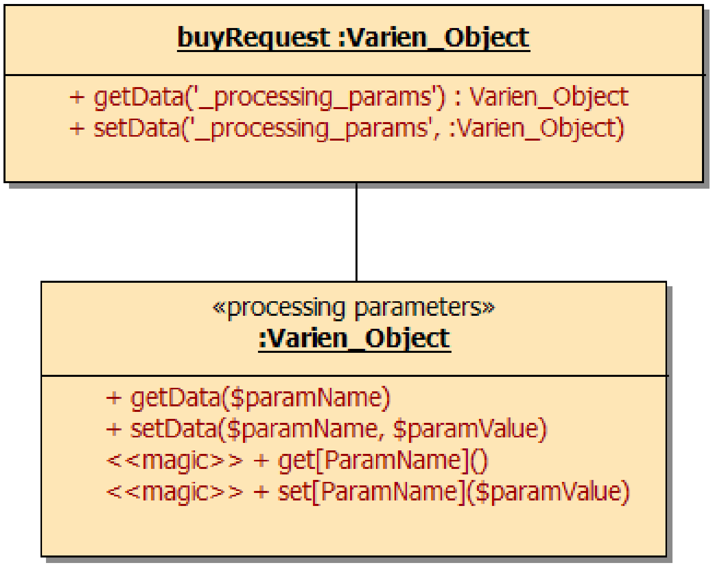

buyRequest :Varien_Object
+ getData('_processing_params') : Varien_Object
+ setData('_processing_params', :Varien_Object)
«processing parameters»
:Varien_Object
+ getData($paramName)
+ setData($paramName, $paramValue)
<<magic>> + get[ParamName]()
<<magic>> + set[ParamName]($paramValue)

Fig. 48. Embedding data into buyRequest

Additional meta-information is embedded via the `Mage_Catalog_Helper_Product::addParamsToBuyRequest($buyRequest, $params)` method. The technique is simple: the provided $params array is added to a *processing parameters object*, residing in the `_processing_params` key inside the buyRequest. Later that object and its data can be retrieved by any configuration processing model. Such an approach allows the transferring of special processing parameters across the system together with the buyRequest without the need to add more parameters to method signatures.

The diagram in Fig. 48 shows that parameters can be added to the *processing params object* through the usual `Varien_Object` interface – the standard `getData()/setData()` and the magic `getParamName()/setParamName()` methods can be used.

Look how custom file option and backend controllers use this feature to transfer data about the name prefix that was used by the Config Manager. First, the backend controller embeds the information about the prefix into the buyRequest:

Mage_Adminhtml_Sales_Order_CreateController

```
/**
 * Process buyRequest file options of items
 *
 * @param array $items
 * @return array
 */
protected function _processFiles($items)
{
    /* @var $productHelper Mage_Catalog_Helper_Product */
    $productHelper = Mage::helper('catalog/product');
    foreach ($items as $id => $item) {
        $buyRequest = new Varien_Object($item);

        $params = array('files_prefix' => 'item_' . $id . '_');
        $buyRequest = $productHelper
            ->addParamsToBuyRequest($buyRequest, $params);

        if ($buyRequest->hasData()) {
            $items[$id] = $buyRequest->toArray();
        }
    }
    return $items;
}
```

After going through many models, this buyRequest gets to the File Custom Option validation process inside its `_validateUploadedFile()` method:

```
Mage_Catalog_Model_Product_Option_Type_File

/**
 * Validate uploaded file
 *
 * @throws Mage_Core_Exception
 * @return Mage_Catalog_Model_Product_Option_Type_File
 */
protected function _validateUploadedFile()
{
    $option = $this->getOption();
    $processingParams = $this->_getProcessingParams();

    $upload   = new Zend_File_Transfer_Adapter_Http();
    $file = $processingParams->getFilesPrefix()
        . 'options_' . $option->getId() . '_file';
    ...
    $fileInfo = $upload->getFileInfo($file);
...
}

/**
 * Returns additional params for processing options
 *
 * @return Varien_Object
 */
protected function _getProcessingParams()
{
    $buyRequest = $this->getRequest();
    $params = $buyRequest->getData('_processing_params');
    ...
    if ($params instanceof Varien_Object) {
        return $params;
    }
    return new Varien_Object();
}
```

As you can see, `_validateUploadedFile()` retrieves the processing parameters via a call to `_getProcessingParams()` and builds the key used to search for the uploaded file. Remember that the same validation models are used both for the frontend and the backend. While the code above will always return an empty prefix for the frontend (there is no Config Manager), for the backend case it might return a meaningful value. Without using this value (the renaming prefix), the validation model would not be able to successfully receive and process the file uploaded as part of the postponed submission.

One more thing embedded as meta-information into the buyRequest (the POSTed new configuration), is a product's previous configuration. Actually, it's just a buyRequest too. This embedded data is intended to allow the validation model to keep the file option's previous value when the shopper edits the configuration and submits the new one without changing the file.

Because files are stored outside the database, it's a best practice to delete the old file when the configuration is changed and a new file is uploaded. Fig. 49 shows what the file upload control in the Admin Panel typically looks like when the store manager has clicked the *Change* link to replace the previously uploaded file with a new one.

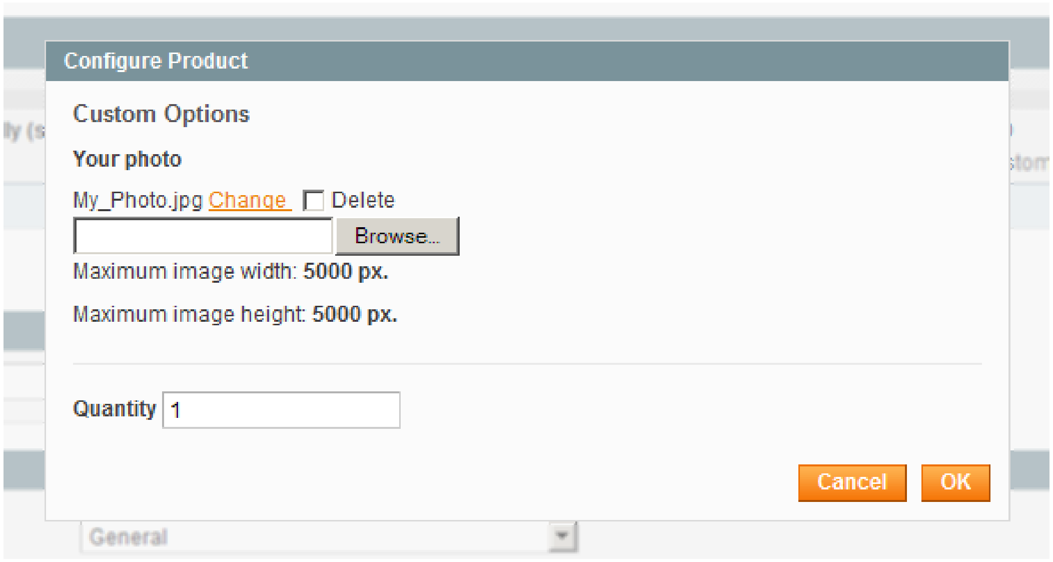


Fig. 49. File upload control

After an item's product configuration is modified, all unchanged inputs are submitted with the previously configured values. File controls are exceptions to this scheme (due to the specifics already described). When their values are not changed by the user, Magento doesn't receive the previous settings in the `$_POST` or `$_FILES` arrays. Thus the system needs to internally load an item's previous configuration and provide it to the option model, which will use the configuration to copy the old file option's value to the new one.

Look how the custom file option and the Wishlist work together to achieve the goal of keeping the old option value. The Wishlist extracts and embeds the current item configuration into the buyRequest in the `current_config` key:

```
Mage_Wishlist_Model_Wishlist

/**
 * Update wishlist Item and set data from request
 *
 * @param int $itemId
 * @param Varien_Object $buyRequest
 * @param null|array|Varien_Object $params
 * @return Mage_Wishlist_Model_Wishlist
 */
public function updateItem($itemId, $buyRequest, $params = null)
{
    $item = $this->getItem((int)$itemId);
    ...

    $params = new Varien_Object($params);
    $params->setCurrentConfig($item->getBuyRequest());
    $buyRequest = Mage::helper('catalog/product')
        ->addParamsToBuyRequest($buyRequest, $params);

    $resultItem = $this->addNewItem($product, $buyRequest, true);
    ...
}
```

After that, the buyRequest travels across the system and comes to the file custom option model. If the user decided not to change the file, the model just copies the previous setting to its value:

```
Mage_Catalog_Model_Product_Option_Type_File

/**
 * Returns file info array if we need to get file from already
existing file.
 * Or returns null, if we need to get file from uploaded array.
 *
 * @return null|array
 */
protected function _getCurrentConfigFileInfo()
{
    $option = $this->getOption();
    $optionId = $option->getId();
    $processingParams = $this->_getProcessingParams();
    $buyRequest = $this->getRequest();

    // Check maybe restore file from config requested
    $optionActionKey = 'options_' . $optionId . '_file_action';
    if ($buyRequest->getData($optionActionKey) == 'save_old') {
        $fileInfo = array();
        $currentConfig = $processingParams->getCurrentConfig();
        if ($currentConfig) {
            $fileInfo = $currentConfig
                ->getData('options/' . $optionId);
        }
        return $fileInfo;
    }
    return null;
}
```

Ok, now you know about embedding processing parameters (we can call them *instructions*) into the buyRequest. Then let's sum it up.

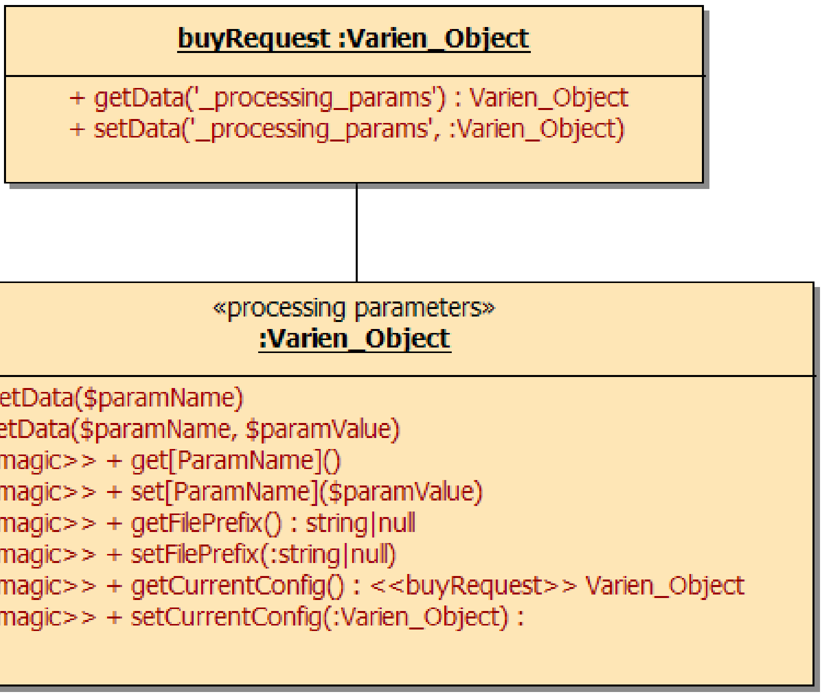

buyRequest :Varien_Object
+ getData('_processing_params') : Varien_Object
+ setData('_processing_params', :Varien_Object)
«processing parameters»
:Varien_Object
+ getData($paramName)
+ setData($paramName, $paramValue)
<<magic>> + get[ParamName]()
<<magic>> + set[ParamName]($paramValue)
<<magic>> + getFilePrefix() : string|null
<<magic>> + setFilePrefix(:string|null)
<<magic>> + getCurrentConfig() : <<buyRequest>> Varien_Object
<<magic>> + setCurrentConfig(:Varien_Object) :

Fig. 50. Embedding file control meta-data in the buyRequest

The redrawn diagram in Fig. 50 shows the two keys mentioned above, currently used by Magento in the processing parameters – `file_prefix` and `current_config` (the corresponding magic methods were added to the diagram, too). However, more processing parameter keys may also be reserved and used by Magento in future.

The `file_prefix` entry contains the prefix added by the Config Manager to the standard names of the file input fields. The `current_config` entry contains the buyRequest with the current configuration of an item that has been reconfigured.

Your configuration options can implement their own scheme for file management. You're not obliged to reproduce the Core's implementation with file prefixes. Just follow the best practice and don't leave trash in the filesystem – i.e., delete files that are no longer needed.

Although the feature of embedding additional processing parameters was developed to support file controls, it's quite abstract. Your own models can set and get any other keys in the embedded parameters. Feel free to use this feature both in the frontend and the backend to transfer more data along with the buyRequest any time you need it.

The Config Manager's Interface

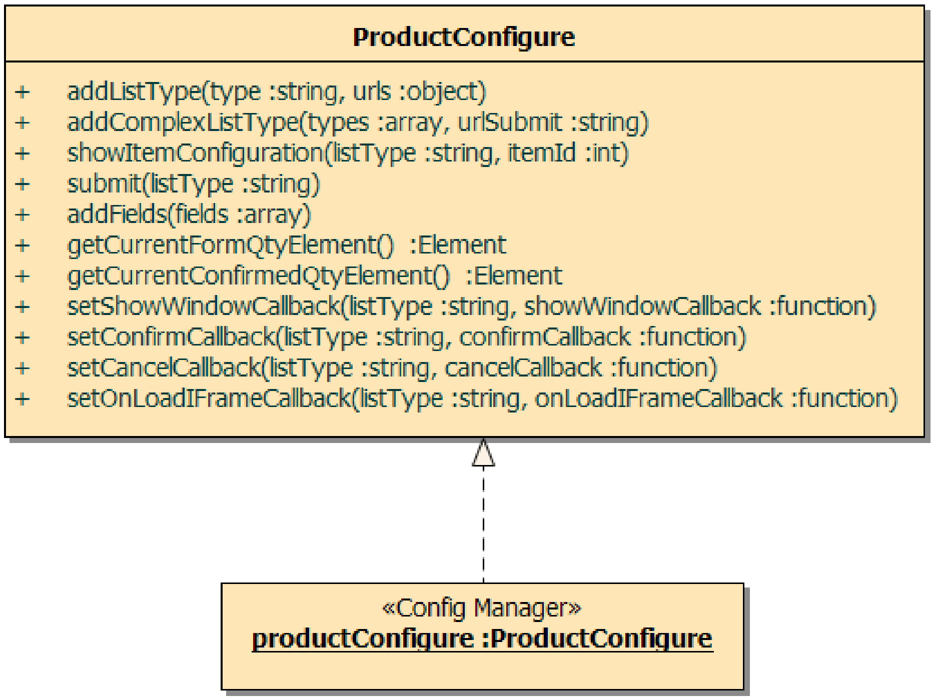

ProductConfigure
+ addListType(type :string, urls :object)
+ addComplexListType(types :array, urlSubmit :string)
+ showItemConfiguration(listType :string, itemId :int)
+ submit(listType :string)
+ addFields(fields :array)
+ getCurrentFormQtyElement() :Element
+ getCurrentConfirmedQtyElement() :Element
+ setShowWindowCallback(listType :string, showWindowCallback :function)
+ setConfirmCallback(listType :string, confirmCallback :function)
+ setCancelCallback(listType :string, cancelCallback :function)
+ setOnLoadIFrameCallback(listType :string, onLoadIFrameCallback :function)
«Config Manager»
productConfigure :ProductConfigure

Fig. 51. Config Manager's structure

When developing your own pages that allow product configuration, you will definitely need to work with the Config Manager. As mentioned earlier, it's represented by the global object `productConfigure`, being an instance of the `ProductConfigure` class. All the communications concerning the configuring of products and saving their configurations happens with the help of this object.

In this chapter I will explain the interface that the Config Manager provides.

The function **`addListType(type, urls)`** is the most frequently used method. It adds a new List and its endpoint urls to the Config Manager. The `type` parameter holds a string – the list identifier (e.g. '`wishlist`', '`rcompared`', etc.) – that must be unique within the Config Manager instance. The `urls` parameter represents a record (a javascript object) that keeps the urls which the Config Manager uses to communicate with the backend list's controller.

Each `urls` record contains the following fields:

- `urlFetch` – the http-path to retrieve the html with the configuration controls. Whenever the shopper wants to configure a product, the Config Manager makes a request to this url, providing the id of the item being configured.
- `urlConfirm` – the http-path to **instantly** submit the item configuration when the shopper clicks the “OK” button in the configuration popup.
- `urlSubmit` – the http-path to use for **postponed** submission. You need to call the Config Manager's `submit()` method to make this kind of submission.

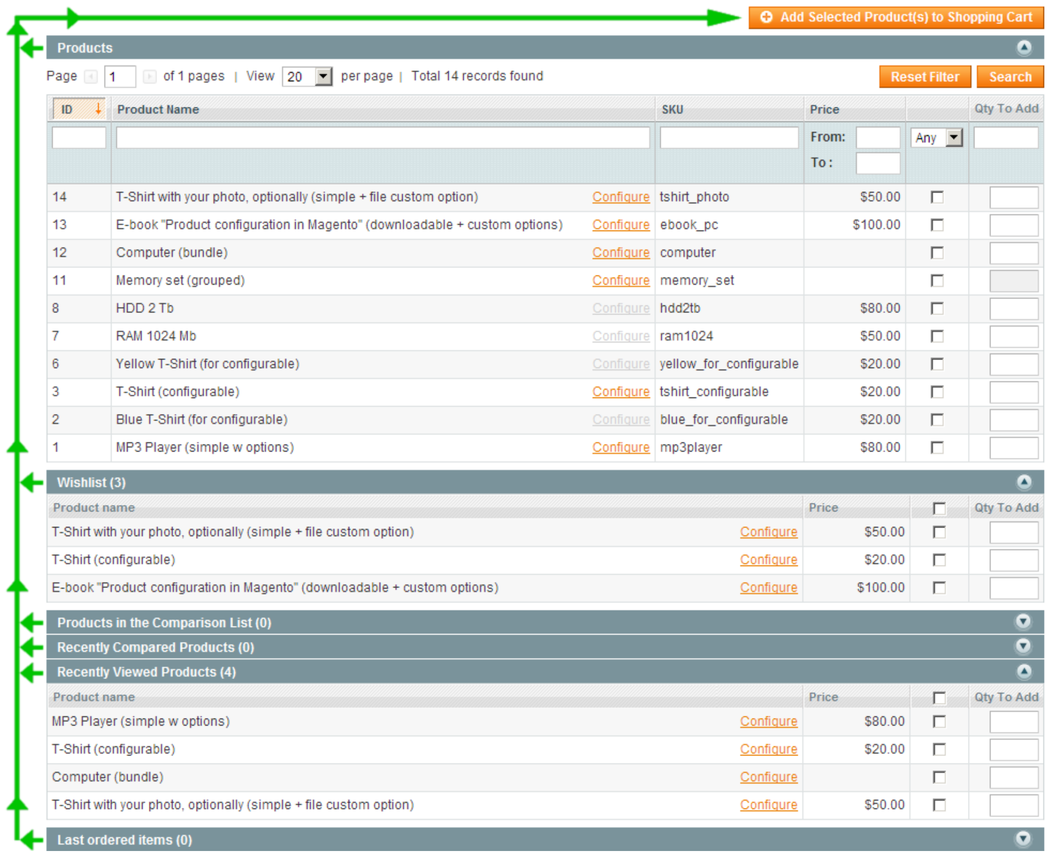
You should always specify the `urlFetch` entry and either `urlConfirm` or `urlSubmit` when adding a list to the Config Manager.

A typical use case for this method, as implemented by the sidebar Wishlist, looks like:

```
productConfigure.addListType(
    'sidebar_wishlist',
    {
        urlFetch: 'http://example.com/admin/customer_wishlist_
product_composite_wishlist/configure/',
        urlConfirm: 'http://example.com/admin/sales_order_create/
addConfigured/'
    }
);
```

The function **`addComplexListType(types, urlSubmit)`** is a rarely used method (in fact, it's needed only once in the Magento core). The purpose of this method is to compose a *complex list*, collecting the identifiers of typical simple lists (they should be added via `addListType()` before calling this method). Later you can submit all of these simple lists in a single AJAX-request, just by calling the Config Manager's `submit()` method and providing the complex list's identifier.

Complex list support is used by the Enterprise Manage Customer page (see Fig. 52), where product configurations from several lists are submitted with the click of a single button. Otherwise, it would be ridiculous to call `submit()` once for every list, issuing a bunch of AJAX-requests and waiting for all of them to complete.



Products

Page 1 of 1 pages | View 20 per page | Total 14 records found

ID	Product Name		SKU	Price		Qty To Add
				From: To :	Any	
14	T-Shirt with your photo, optionally (simple + file custom option)	Configure	tshirt_photo	$50.00	☐	
13	E-book "Product configuration in Magento" (downloadable + custom options)	Configure	ebook_pc	$100.00	☐	
12	Computer (bundle)	Configure	computer		☐	
11	Memory set (grouped)	Configure	memory_set		☐	
8	HDD 2 Tb	Configure	hdd2tb	$80.00	☐	
7	RAM 1024 Mb	Configure	ram1024	$50.00	☐	
6	Yellow T-Shirt (for configurable)	Configure	yellow_for_configurable	$20.00	☐	
3	T-Shirt (configurable)	Configure	tshirt_configurable	$20.00	☐	
2	Blue T-Shirt (for configurable)	Configure	blue_for_configurable	$20.00	☐	
1	MP3 Player (simple w options)	Configure	mp3player	$80.00	☐	

Wishlist (3)

Product name		Price	☐	Qty To Add
T-Shirt with your photo, optionally (simple + file custom option)	Configure	$50.00	☐	
T-Shirt (configurable)	Configure	$20.00	☐	
E-book "Product configuration in Magento" (downloadable + custom options)	Configure	$100.00	☐	

Products in the Comparison List (0)

Recently Compared Products (0)

Recently Viewed Products (4)

Product name		Price	☐	Qty To Add
MP3 Player (simple w options)	Configure	$80.00	☐	
T-Shirt (configurable)	Configure	$20.00	☐	
Computer (bundle)	Configure		☐	
T-Shirt with your photo, optionally (simple + file custom option)	Configure	$50.00	☐	

Last ordered items (0)

Fig. 52. Complex list, consisting of simple lists

The `addComplexListType()` method returns an auto-generated id for the newly-created complex list. You can pass this id to the `submit()` method, and the Config Manager will issue the AJAX POST-request containing the products (and their configurations) from all the lists contained within the complex list.

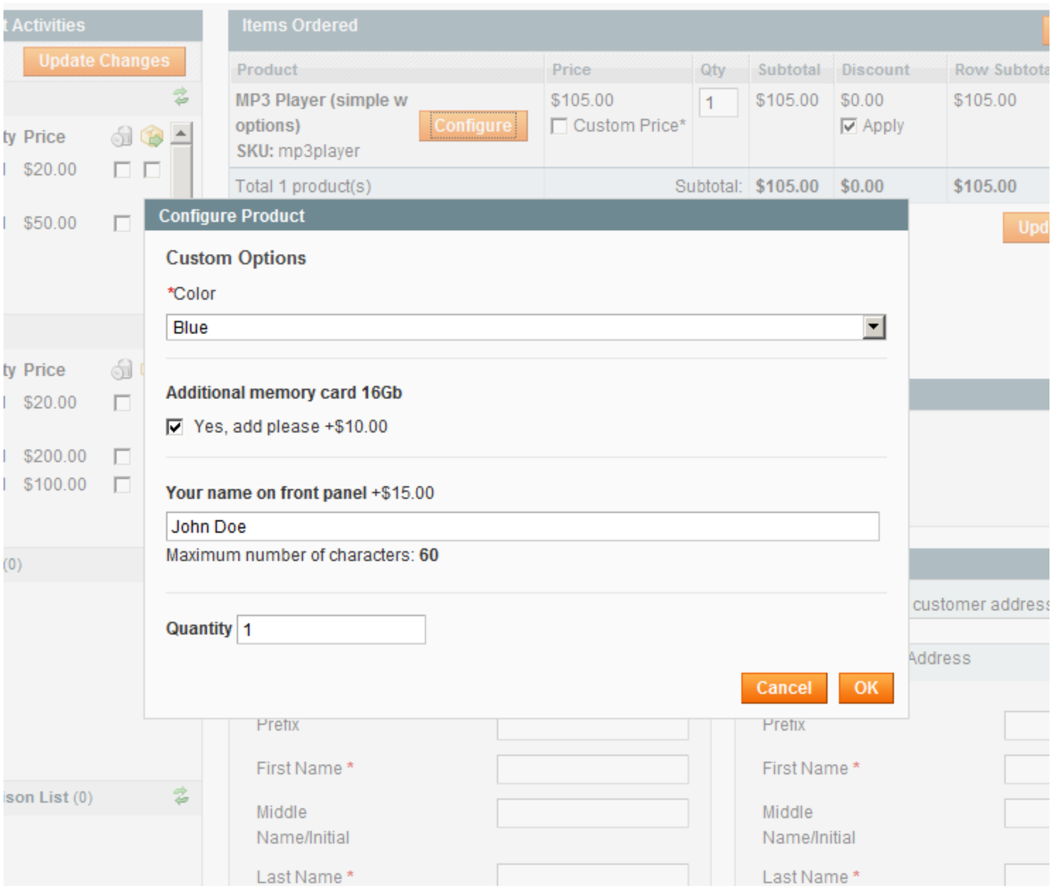
A typical usage (reformatted a little for better understanding) of this method by the Enterprise Manage Customer page looks like:

```
var types = ['product_to_add', 'wishlist', 'compared', 'ordered',
    'rviewed', 'rcompared'];
var url = '/admin/checkout/addToShoppingCart';
var listType = productConfigure.addComplexListType(types, url);
productConfigure.submit(listType);
```

In this code snippet the Config Manager is instructed to submit all modified product configurations from six different lists in a single AJAX request.

The function **`showItemConfiguration(listType, itemId)`** tells the Config Manager to make the AJAX-request to the backend, retrieve the list item configuration and show it as popup dialog.

This method is typically used by the Create Order page, when the admin clicks the *Configure* button in the *Items Ordered* block. The Javascript event observer receives this click and calls the Config Manager's `showItemConfiguration()` method. The rest is done by the Config Manager – it loads the product configuration from the server and shows a visual popup with the configuration controls (Fig. 53).



Product	Price	Qty	Subtotal	Discount	Row Subtotal
MP3 Player (simple w options) SKU: mp3player	$105.00 ☐ Custom Price*	1	$105.00	$0.00 ☑ Apply	$105.00
Total 1 product(s)	Subtotal:		$105.00	$0.00	$105.00

ty Price	
$20.00	☐ ☐
$50.00	☐

ty Price	
$20.00	☐
$200.00	☐
$100.00	☐

Configure Product

Custom Options

*Color: Blue

Additional memory card 16Gb

☑ Yes, add please +$10.00

Your name on front panel +$15.00: John Doe

Maximum number of characters: 60

Quantity: 1

Cancel OK

Fig. 53. Configuration popup, shown as the result of a call to the `showItemConfiguration()` method

The actual HTML code of the *Configure* button, shown in Fig. 53, looks like

```
<button onclick="order.showQuoteItemConfiguration(ITEM_ID)">
    <span>Configure</span>
</button>
```

Clicking the button invokes the custom code of the `order` Javascript object, which manages the user interface of the Create Order page. The object makes some preparations and calls the Config Manager's `showItemConfiguration()` method:

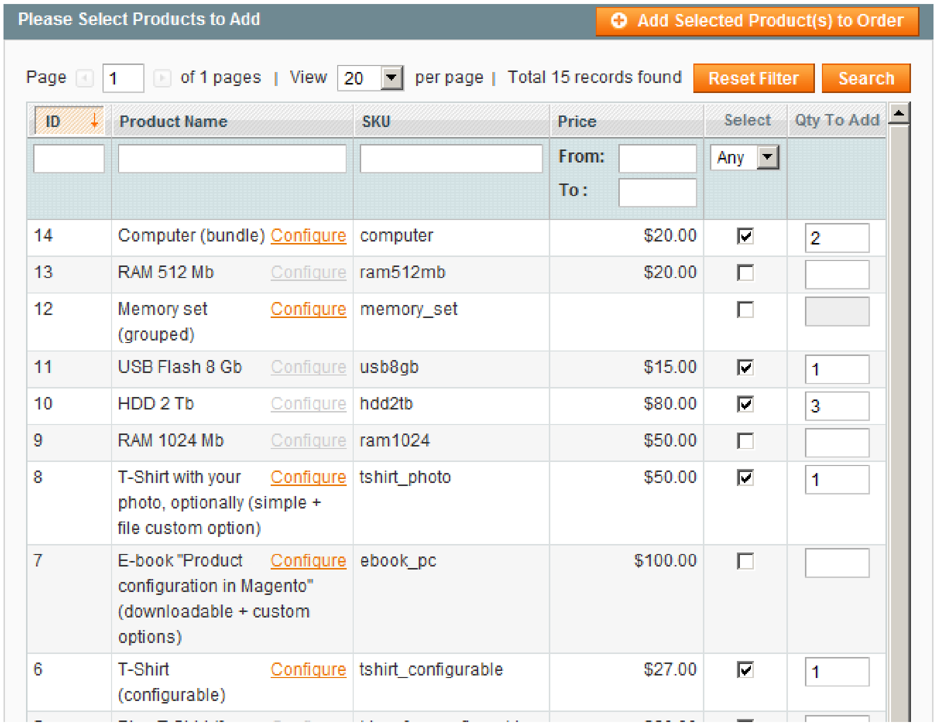
```
/js/mage/adminhtml/sales.js

/**
 * Show configuration of quote item
 *
 * @param itemId
 */
showQuoteItemConfiguration: function(itemId){
    var listType = 'quote_items';
    ...
    productConfigure.showItemConfiguration(listType, itemId);
}
```

The function **`submit(listType)`** does what its name says – it submits all the products configured within the specified list. This routine is used for lists with postponed submissions and should be invoked when the admin clicks the appropriate button (such as “Update Changes”). The `listType` is a list identifier, which can be either a simple list or a complex list. In the latter case all the simple lists, composing the complex one, are submitted within a single AJAX request.

A typical usage is on the Create Order page, where an admin chooses products to be added to the order and clicks the “Add Selected Product(s) to Order” button. Visually, before submitting, the page looks like Fig. 54.

Clicking this button calls `submit()` with the identifier of the list being submitted:

```
productConfigure.submit('add_to_order');
```



Please Select Products to Add — Add Selected Product(s) to Order

Page 1 of 1 pages | View 20 per page | Total 15 records found — Reset Filter — Search

ID	Product Name		SKU	Price	Select	Qty To Add
				From: / To :	Any	
14	Computer (bundle)	Configure	computer	$20.00	✓	2
13	RAM 512 Mb	Configure	ram512mb	$20.00	☐	
12	Memory set (grouped)	Configure	memory_set		☐	
11	USB Flash 8 Gb	Configure	usb8gb	$15.00	✓	1
10	HDD 2 Tb	Configure	hdd2tb	$80.00	✓	3
9	RAM 1024 Mb	Configure	ram1024	$50.00	☐	
8	T-Shirt with your photo, optionally (simple + file custom option)	Configure	tshirt_photo	$50.00	✓	1
7	E-book "Product configuration in Magento" (downloadable + custom options)	Configure	ebook_pc	$100.00	☐	
6	T-Shirt (configurable)	Configure	tshirt_configurable	$27.00	✓	1

Fig. 54. Products and their configurations are prepared to be added to an order

---

The very useful `addFields(fields)` method allows you to add your own custom data to the list's submit request (made by the Config Manager).

Look how the Create Order page passes additional flags when submitting the products list. These flags have nothing to do with product configuration, but it's convenient to use the Config Manager's request as a transport and send these flags to a controller:

/js/mage/adminhtml/sales.js

```
productConfigureSubmit : function(listType, area, fieldsPrepare,
    itemsFilter) {
    ...
    // prepare additional fields
    fieldsPrepare = this.prepareParams(fieldsPrepare);
    fieldsPrepare.reset_shipping = 1;
    fieldsPrepare.json = 1;

    // create fields
    var fields = [];
    for (var name in fieldsPrepare) {
        fields.push(new Element('input',
            {
                type: 'hidden',
                name: name,
                value: fieldsPrepare[name]
            }
        ));
    }
    productConfigure.addFields(fields);
    ...
    productConfigure.submit(listType);
}
```

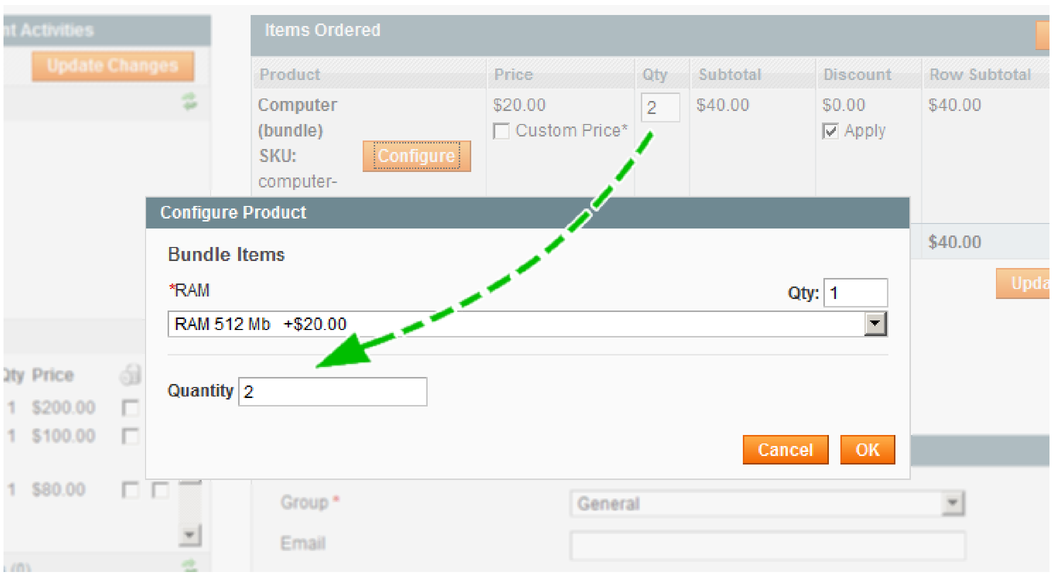
Without this feature, we would need to issue a second AJAX-request to pass these additional parameters. The attached data is meant for a processing controller. Additional parameters instruct the controller to execute some specific logic. They are not related to product configuration and won't be present in the items' buyRequests.

The explicit example above shows that we instruct the controller for the Create Order page to reset the shipping options block and return the data in JSON format, after the list with the product configurations is processed.

The **`getCurrentFormQtyElement()`** and **`getCurrentConfirmedQtyElement()`** methods come as a pair. The first one retrieves the Qty input in the visible configuration popup, the second retrieves it after the user closes the popup. These methods are used to synchronize the qty input in the items grid and the qty input in the configuration popup.

Fig. 55 shows how data from the *Items Ordered* grid is copied to the configuration popup. And here is the actual code of this copying operation, which is executed via event callbacks:

```
/js/mage/adminhtml/sales.js

/**

 * Show configuration of quote item
 *
 * @param itemId
 */
showQuoteItemConfiguration: function(itemId) {
    var listType = 'quote_items';
    var qtyElement = $('order-items_grid')
        .select('input[name="item\['+itemId+'\]\[qty\]"]')[0];

    productConfigure.setShowWindowCallback(listType, function() {
        // sync qty of grid and qty of popup
        var formCurrentQty = productConfigure.
            getCurrentFormQtyElement();
        if (formCurrentQty && qtyElement
            && !isNaN(qtyElement.value))
        {
            formCurrentQty.value = qtyElement.value;
        }
    }.bind(this));

    productConfigure.setConfirmCallback(listType, function() {
        // sync qty of popup and qty of grid
        var confirmedCurrentQty = productConfigure.
            getCurrentConfirmedQtyElement();
        if (qtyElement && confirmedCurrentQty &&
            !isNaN(confirmedCurrentQty.value))
        {
            qtyElement.value = confirmedCurrentQty.value;
        }
    }.bind(this));

    productConfigure.showItemConfiguration(listType, itemId);
}
```



Product	Price	Qty	Subtotal	Discount	Row Subtotal
Computer (bundle) SKU: computer-	$20.00 ☐ Custom Price*	2	$40.00	$0.00 ☑ Apply	$40.00

Bundle Items

*RAM Qty: 1

RAM 512 Mb +$20.00

Quantity 2

Fig. 55. Quantity inputs synchronized

I have already used the body of the `showQuoteItemConfiguration()` method in an example above. This method is responsible for showing the configuration popup when a user clicks the *Configure* button. This time I replaced the “some preparations” comment with real code which shows that two popup window events are observed. On *popup show* we find the quantity field and populate it with the quantity value from the grid. On *popup hide* we get the quantity value entered in the popup and send it back to the grid.

There are three methods to set callbacks and receive notifications during popup show/hide actions:

- **`setShowWindowCallback(listType, showWindowCallback)`** commands the Config Manager to call the `showWindowCallback` function when the item configuration controls are loaded from the backend and the popup is shown.
- **`setConfirmCallback(listType, confirmCallback)`** commands the Config Manager to call the confirmCallback function after the shopper clicks the “OK” button.
- **`setCancelCallback(listType, cancelCallback)`** commands the Config Manager to call the cancelCallback function after the shopper clicked the “Cancel” button.

Setting your own callbacks allows you to add more Javascript logic to the processing of user actions.

The **`setOnLoadIFrameCallback(listType, onLoadIFrameCallback)`** method is one more routine to control the configuration process flow. It instructs the Config Manager to call `onLoadIFrameCallback(response)` after the submission result has been received from the backend. The callback's `response` parameter contains the JSON returned by the server.

The Create Order page uses this callback to refresh the different areas within the page, after a product has been added to the order:

/js/mage/adminhtml/sales.js

```
/**
 * Show configuration of product and add handlers on submit form
 *
 * @param productId
 */
sidebarConfigureProduct: function (listType, productId, itemId) {
    ...
    // response handler
    productConfigure.setOnLoadIFrameCallback(
        listType,
        function(response) {
            if (!response.ok) {
                return;
            }
            this.loadArea(
                ['items',
                'shipping_method',
                'billing_method','totals',
                'giftmessage'],
                true
            );
        }.bind(this)
    );
    ...
}
```

That's everything about the Config Manager's interface – now you're ready to organize your own Admin Panel pages that work with product configurations.

AJAX-Requests Made by the Config Manager

We have already talked about client-side programming – the Config Manager and its integration with the other Javascript on the page. Now let's have a look at the backend processes – how the requests from the Config Manager are handled.

There are two types of requests coming to the backend related to items in the lists. They are *fetch configuration html* and *process submitted configuration(s)*. The first one is used by the Config Manager to retrieve html with configuration controls. This type of request is made to the list's `urlFetch` endpoint url. The *process submitted configuration(s)* request is issued by the Config Manager to the list's `urlConfirm` or `urlSubmit` endpoint url.

The request with the submitted item's product settings is way too simple – it has the standard logic for updating the configuration of a product. There the controller action just loads the item and updates it using the POSTed data.

The fetch configuration html request is much more interesting. Let's discuss it in detail.

Processing *fetch configuration html* Requests

The logic of processing this kind of request is divided between the controller action and a special helper.

The controller receives the id of an item to be configured, and is responsible for loading it together with its product and configuration. This is the part you will need to develop, if you're building some grids with composite products.

The special helper `Mage_Adminhtml_Helper_Catalog_Product_Composite` takes further responsibility for rendering configuration controls and returning them to the script. To make it work you need to pass the required information (described below) from the controller action to this helper. The rest of the work on composing the actual html with configuration controls will be done automatically – you don't need to implement anything.

The code below shows the implementation of the controller's action, which returns the configuration html for the product in a Quote object:

Mage_Adminhtml_Sales_Order_CreateController

```
/*
 * Ajax handler to response configuration fieldset of composite
 * product in quote items
 *
 * @return Mage_Adminhtml_Sales_Order_CreateController
 */
public function configureQuoteItemsAction()
{
    // Prepare data
    $configureResult = new Varien_Object();
    try {
        $quoteItemId = (int) $this->getRequest()->getParam('id');
        if (!$quoteItemId) {
            Mage::throwException(
                $this->__('Quote item id is not received.')
            );
        }

        $quoteItem = Mage::getModel('sales/quote_item')
            ->load($quoteItemId);
        if (!$quoteItem->getId()) {
            Mage::throwException(
                $this->__('Quote item is not loaded.')
            );
        }

        $configureResult->setOk(true);

        $configureResult->setBuyRequest(
            $quoteItem->getBuyRequest()
        );
        $configureResult->setCurrentStoreId(
            $quoteItem->getStoreId()
        );
        $configureResult->setProductId($quoteItem->getProductId());
        $sessionQuote = Mage::getSingleton(
            'adminhtml/session_quote'
        );
        $configureResult->setCurrentCustomerId(
            $sessionQuote->getCustomerId()
    );
    } catch (Exception $e) {
        $configureResult->setError(true);
        $configureResult->setMessage($e->getMessage());
    }

    // Render page
    /*@var $helper Mage_Adminhtml_Helper_Catalog_Product_Composite*/
    $helper = Mage::helper('adminhtml/catalog_product_composite');
    $helper->renderConfigureResult($this, $configureResult);

    return $this;
}
```

As you can see, the controller makes all the required validations and composes a `$configureResult` instance of `Varien_Object` with predefined fields. The helper does the rest of the hard work to render the response and output it in a format which is understood by the Config Manager on the client side.

The configureResult's fields depend on whether the operation is successful or not. If everything is fine with the request (the user has enough permissions, the item was loaded, no errors occurred), then configureResult will have the `ok` field set to `TRUE`. The rest of the required fields are self-explanatory: `product_id`, `buy_request`, `current_store_id` and `current_customer_id`.

If there's a problem serving the request, the `error` field is set to `TRUE` and the `message` field will contain a human-readable error message to display on the client-side.

Adding Configuration Controls to the Popup

Although the Composite helper renders the popup html on its own, you should know how to add more controls to it. Whether you're developing your own product type, or just adding some configuration controls to already existing products, this chapter will help you embed your inputs into the configuration popup.

This chapter requires that you understand the layout-block-template Magento architecture. If you aren’t familiar with this you’d better study it, starting with the Intro to Layouts (http://www.magentocommerce.com/design_guide/articles/intro-to-layouts) article, and then return to this guide.

The method that Magento uses to dynamically compose a popup html is the same one the Catalog module uses to compose the View Product page. The magic happens in the layout files – whenever a page is loaded, a standard set of layout handles are added. In addition to these handles, the helper makes an attempt to load a specific product type dependent handle.

Look at the helper's code:

```
Mage_Adminhtml_Helper_Catalog_Product_Composite

 /**
 * Init composite product configuration layout
 *
 * $isOk - true or false, whether action was completed nicely or
 * with some error
 * If $isOk is FALSE (some error during configuration), so
 * $productType must be null
 *
 * @param Mage_Adminhtml_Controller_Action $controller
 * @param bool $isOk
 * @param string $productType
 * @return Mage_Adminhtml_Helper_Catalog_Product_Composite
 */
protected function _initConfigureResultLayout($controller, $isOk,
    $productType)
{
    $update = $controller->getLayout()->getUpdate();
    if ($isOk) {
        $update->addHandle(
            'ADMINHTML_CATALOG_PRODUCT_COMPOSITE_CONFIGURE'
        )
        ->addHandle('PRODUCT_TYPE_' . $productType);
    } else {
        $update->addHandle(
            'ADMINHTML_CATALOG_PRODUCT_COMPOSITE_CONFIGURE_ERROR'
        );
    }
    $controller->loadLayoutUpdates()->generateLayoutXml()
        ->generateLayoutBlocks();
    return $this;
}
```

There are two update handles loaded – the standard ADMINHTML_CATALOG_PRODUCT_COMPOSITE_CONFIGURE handle and the dynamic PRODUCT_TYPE_$productType handle. Notice the handle names are uppercase – this is done so that they will never collide with controller action handles.

If you check the Adminhtml layout file, you will see that the standard handle adds blocks to render the custom options and the qty input:

```
/app/design/adminhtml/default/default/layout/catalog.xml

<ADMINHTML_CATALOG_PRODUCT_COMPOSITE_CONFIGURE>
```

```
    <block type="adminhtml/catalog_product_composite_fieldset"
        name="product.composite.fieldset">
        <block name="product.composite.fieldset.options"
             type="adminhtml/catalog_product_composite_fieldset_
             options" template="catalog/product/composite/fieldset/
             options.phtml">
            ...
            <block name="product.composite.fieldset.options.js"
               type="core/template"
               template="catalog/product/composite/fieldset/options/
               js.phtml"/>
        </block>
        <block name="product.composite.fieldset.qty"
            type="adminhtml/catalog_product_composite_fieldset_qty"
            template="catalog/product/composite/fieldset/qty.
            phtml"/>
    </block>
</ADMINHTML_CATALOG_PRODUCT_COMPOSITE_CONFIGURE>
```

The addition of the generic `PRODUCT_TYPE_$productType` handle allows the configuration popup to be dynamically extended by new modules. Each product type can insert its own blocks into the rendered html. That's how the Bundle product does it via its layout file:

```
/app/design/adminhtml/default/default/layout/bundle.xml

<PRODUCT_TYPE_bundle>
    <reference name="product.composite.fieldset">
        <block type="bundle/adminhtml_catalog_product_composite_
            fieldset_bundle"
            before="product.composite.fieldset.options"
            name="product.composite.fieldset.bundle"
            template="bundle/product/composite/fieldset/options/
            bundle.phtml">
            ...
        </block>
    </reference>
</PRODUCT_TYPE_bundle>
```

So if you've developed your own custom product type, add the `PRODUCT_TYPE_$productType` handle to your Adminhtml layout and implement the rendering block. If you want to change the default rendering for product controls in the popup, use standard techniques to work with handles, blocks and templates.

Visual Controls Inheritance Scheme

Composing configuration html requires Magento to render product configuration controls in the admin area. In fact, all the required controls are already implemented – the shopper uses them on the frontend View Product page. Because it's a bad practice to duplicate the code base, the Core team avoided doing so by creating blocks and models which inherit the frontend ones.

These blocks hardly differ from their frontend ancestors. Just a few methods and templates were changed for the pages, where they had a design unsuitable for the Admin Panel. Also some Javascripts were reimplemented to be compatible with the specifics implied by the Config Manager – this was described earlier, while talking about postponed submissions (unique node ids, Prototype `evalScripts()` and so on).

Let's take the Bundle option inheritance scheme as an example of code reusage:

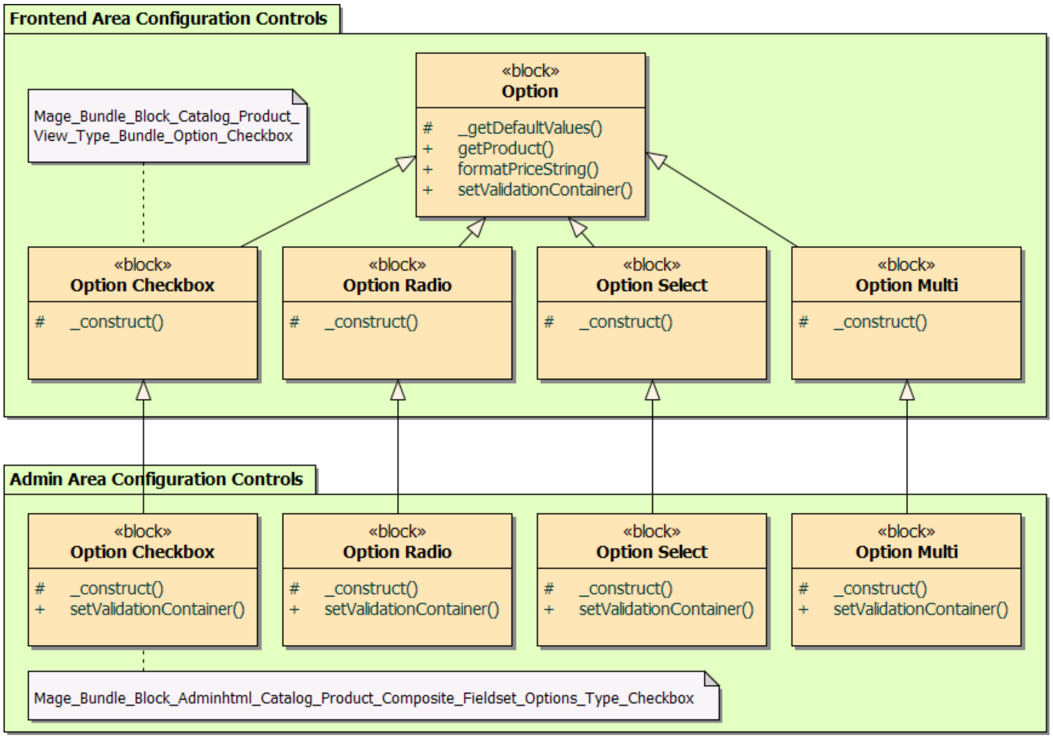


Fig. 56. Bundle option blocks inheritance

Fig. 56 shows everything I said before – the admin controls descend from the frontend controls, and some methods are overwritten.

If you want to avoid legacy code support nightmares, use the same scheme. It allows configuration control logic to be written once, and then used in both the frontend and the Admin Panel. Therefore you should only worry about the implementation of two things – Admin Panel theme design and safe Javascript.

Allowing a Custom Product Type to be Ordered in the Backend

The Create Order page is very important for the store manager. It provides the ability to serve shoppers quickly, within a comfortable environment. Magento cares about this page – it should always be stable and working. That's why **all new product types are not used by the Create Order page by default**.

When your product type is ready to be ordered through the backend (when you have implemented its configuration controls block for the popup), then just add its type code (*type id*) to the `/config/adminhtml/sales/order/create/available_product_types/` node in the `config.xml` file of your module. After that Magento will start showing products of this new product type on the Create Order page.

Take a look at the Bundle config snippet that does this:

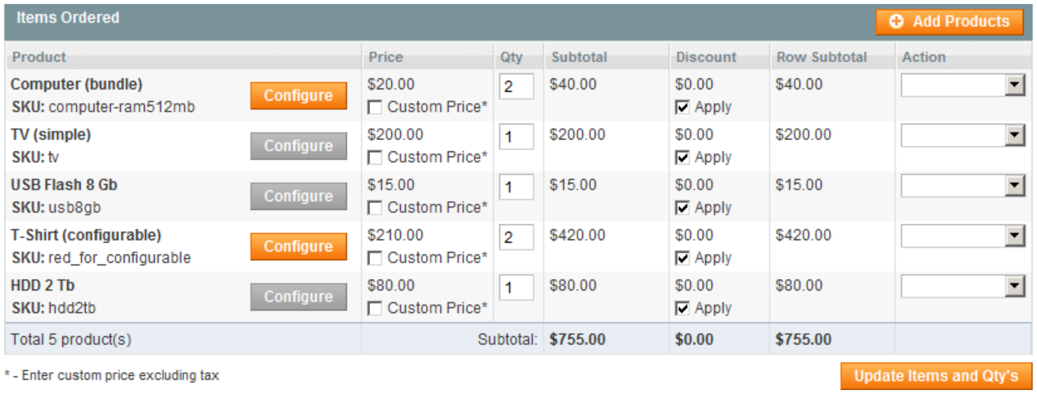
```
/app/code/core/Mage/Bundle/etc/config.xml

<config>
    ...
    <adminhtml>
        ...
        <sales>
            <order>
                <create>
                    <available_product_types>
                        <bundle/>
                    </available_product_types>
                </create>
            </order>
        </sales>
        ...
    </adminhtml>
    ...
</config>
```

I've highlighted the `<bundle>` node – this node name must exactly match the *type id* of the new product you're adding to the system.

Enabling the *Configure* Button

If you look at the lists on the Create Order page, you can notice that some products have the *Configure* buttons (or links) enabled, while others don't (Fig. 57). How does Magento find out whether the products should have configuration controls or not? There is a special method developed to answer this question – see Fig. 58.

Magento calls the product's `canConfigure()` method and receives a boolean result, indicating whether or not the product has visual configuration controls. The product itself returns `TRUE` when it has custom options set up by the admin. Otherwise the product asks its product type instance – because the type instance has additional knowledge on the product's behavior and the presence of visual configuration controls in it.



Items Ordered

Add Products

Product		Price	Qty	Subtotal	Discount	Row Subtotal	Action
Computer (bundle) **SKU:** computer-ram512mb	Configure	$20.00 ☐ Custom Price*	2	$40.00	$0.00 ☑ Apply	$40.00	
TV (simple) **SKU:** tv	Configure	$200.00 ☐ Custom Price*	1	$200.00	$0.00 ☑ Apply	$200.00	
USB Flash 8 Gb **SKU:** usb8gb	Configure	$15.00 ☐ Custom Price*	1	$15.00	$0.00 ☑ Apply	$15.00	
T-Shirt (configurable) **SKU:** red_for_configurable	Configure	$210.00 ☐ Custom Price*	2	$420.00	$0.00 ☑ Apply	$420.00	
HDD 2 Tb **SKU:** hdd2tb	Configure	$80.00 ☐ Custom Price*	1	$80.00	$0.00 ☑ Apply	$80.00	
Total 5 product(s)		Subtotal:		**$755.00**	**$0.00**	**$755.00**	

* - Enter custom price excluding tax

Update Items and Qty's

Fig. 57. Enabled and disabled Configure controls

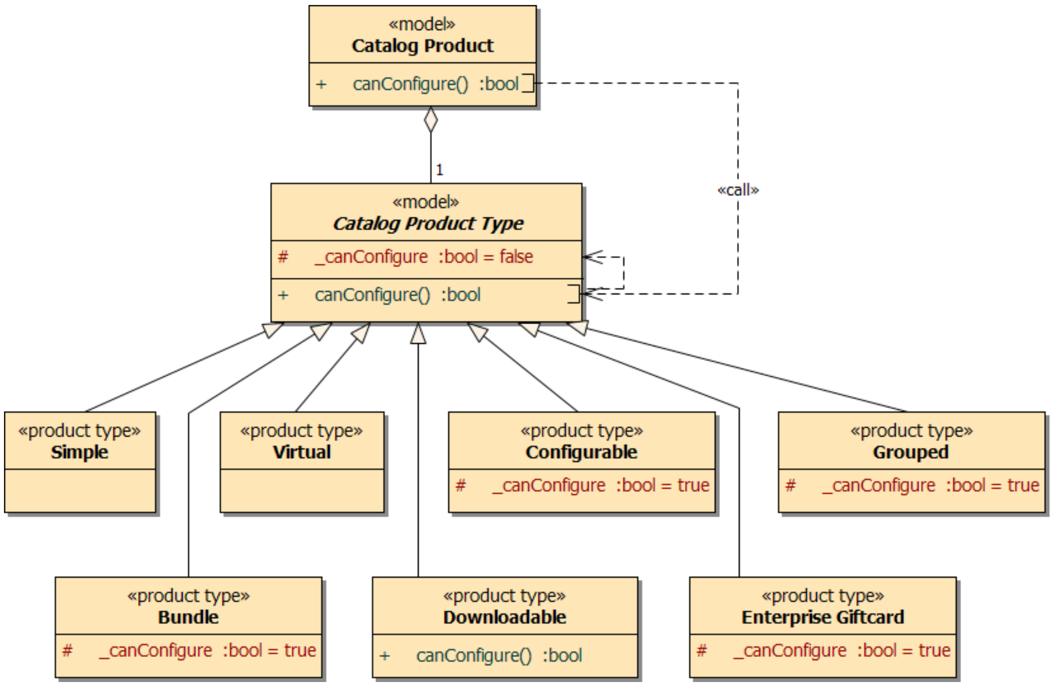


Fig. 58. Checking whether a product has configuration controls

The Abstract class of product type instances defines the public method `canConfigure()` and the protected property `_canConfigure`. The default implementation of the method just returns the `_canConfigure` property. However, specific implementations of different product types can override the property and/or the method. The property should be overridden and set to `TRUE` when the nature of the particular product type implies that it **always** shows some configuration controls. The method should be overridden when the configuration controls are shown (thus the product can be configured) **depending** on the settings that the store manager specified in the backend.

We can see both these cases present in the Core modules. The product types *Simple* and *Virtual* do nothing with the above mentioned property and method. These product types' ability to be configured depends only on the presence of product custom options. And that logic is held by the Product Model on its own, without any help from a Product Type instance. But such product types as *Configurable*, *Grouped*, *Bundle* and *Giftcard* (present in Magento EE) set the default value of the `_canConfigure` property to `TRUE`. All of these product types always have some configuration controls, thus it's enough to set the static value for the `_canConfigure` property. The *Downloadable* product type overrides the method `canConfigure()`, because this product type can be configured only when the store manager has attached a list of additional downloadable files to the product, and the customer is allowed to choose several of them to be downloaded.

For a better understanding of this concept, let's check the implementation of the *Downloadable* product type:

```
Mage_Downloadable_Model_Product_Type

/**
 * Check if downloadable product has links and they can be purchased
 * separately
 *
 * @param Mage_Catalog_Model_Product $product
 * @return bool
 */
public function canConfigure($product = null)
{
    return $this->hasLinks($product)
        && $this->getProduct($product)
            ->getLinksPurchasedSeparately();
}
```

The Downloadable product type model knows about the specifics of this product type. Usually it doesn't differ from other product types and has configuration controls only when custom options are added for the product in the Admin Panel. But when the store manager sets more downloadable links and allows them to be purchased separately, this product starts showing checkboxes to a shopper so the desired links can be selected. In this case the product type model returns `TRUE` as a result of its `canConfigure()` method, so blocks in the Admin Panel know that there are things to configure in this product. Thus the *Configure* button must be enabled.

When developing your own custom product type with additional configuration controls, you need to adjust either the `_canConfigure` property or the `canConfigure()` method in your product type model. If your product type has no special configuration controls, then you don't need to do anything.

Changes on the Manage Customer Page

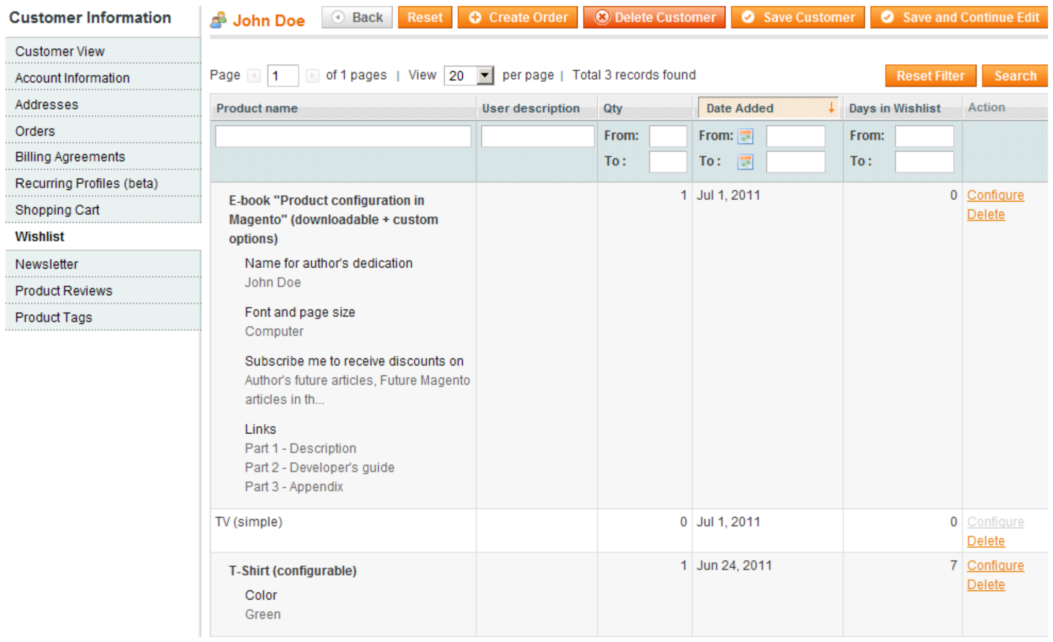
We have talked a lot about the Create Order page, because it's the primary page in the backend that's affected by the Composite Products functionality. But there are several pages in the backend which also feature some configuration specifics and were updated in Magento 1.5. They are:

- Manage Customer page – Wishlist and Shopping Cart tabs
- Manage Customer Shopping Cart page (Magento Enterprise Edition)

There isn't a lot to discuss about these pages, as the whole Composite Products implementation in the backend was explained above. We'll just mention a few specifics for these pages.

Wishlist and Shopping Cart Tabs

The Manage Customer page has two blocks that allow the admin to work with Composite Products – they are the *Wishlist* and *Shopping Cart* tabs. Both are very similar, representing a grid (list) of the products that the customer has added to his Wishlist or Shopping Cart. Both allow reconfiguring products and manipulating them. Because they are alike, we will discuss the Wishlist tab only, and mostly everything said will be applicable to the Shopping Cart as well.



Page 1 of 1 pages | View 20 per page | Total 3 records found

Product name	User description	Qty	Date Added	Days in Wishlist	Action
		From: To:	From: To:	From: To:	
E-book "Product configuration in Magento" (downloadable + custom options) Name for author's dedication John Doe Font and page size Computer Subscribe me to receive discounts on Author's future articles, Future Magento articles in th... Links Part 1 - Description Part 2 - Developer's guide Part 3 - Appendix		1	Jul 1, 2011	0	Configure Delete
TV (simple)		0	Jul 1, 2011	0	Configure Delete
T-Shirt (configurable) Color Green		1	Jun 24, 2011	7	Configure Delete

Fig. 59. Customer's Wishlist in Admin Panel

The Wishlist tab (Fig. 59) has the same internal Javascript as the Create Order page.

The same JS-files are included, the same Config Manager is at work. The only difference is there is only one list – the Wishlist – which features instant configuration submission. All these concepts have already been discussed.

The only interesting thing for us here is the ability of the Wishlist to render the item's configuration options. It's convenient for the store manager, as he is able to see at a glance how the shopper configured the Wishlist items.

Check the code of the template that outputs the grid's cell with the item name and configuration:

/app/design/adminhtml/default/default/template/customer/edit/tab/view/grid/item.phtml

```
<?php
    $product = $this->getProduct();
    $options = $this->getOptionList();
?>

<?php if (!$options): ?>
    <?php echo $this->escapeHtml($product->getName())?>
<?php else: ?>
<div class="product-options">
    <strong>
        <?php echo $this->escapeHtml($product->getName())?>
    </strong>
    <dl>
        <?php foreach ($options as $option) :  ?>
            <dt>
                <?php echo $this->escapeHtml($option['label']) ?>
            </dt>
            <?php $formatedOptionValue = $this
                ->getFormattedOptionValue($option) ;
            ?>
            <dd><?php echo $formatedOptionValue['value'] ?></dd>
        <?php endforeach; ?>
    </dl>
</div>
<?php endif; ?>
```

The array of options is iterated, and the `label` and `value` entries of each option are shown. Sounds familiar? I'm sure you won't be surprised to know that Magento follows its best practice of code reusage here, and uses the Configuration Helpers to retrieve the product options.

Check the `getOptionList()` implementation in the template's block:

Mage_Adminhtml_Block_Customer_Edit_Tab_View_Grid_Renderer_Item

```
/**
```

```
 * Returns list of options and their values for product
 * configuration
 *
 * @return array
 */
protected function getOptionList()
{
    $item = $this->getItem();
    $product = $item->getProduct();
    $helper = $this->_getProductHelper($product);
    return $helper->getOptions($item);
}
```

Exactly the standard scheme of configuration helper use.

Why did I tell you all that information about helpers we already discussed? Two reasons:

1. Now you see it's very easy to implement your own grids that show item configurations. Ten lines of code!
2. Once you've created the configuration helper for your custom product type, it's just a matter of a few layout changes to give it to all the Magento core blocks.

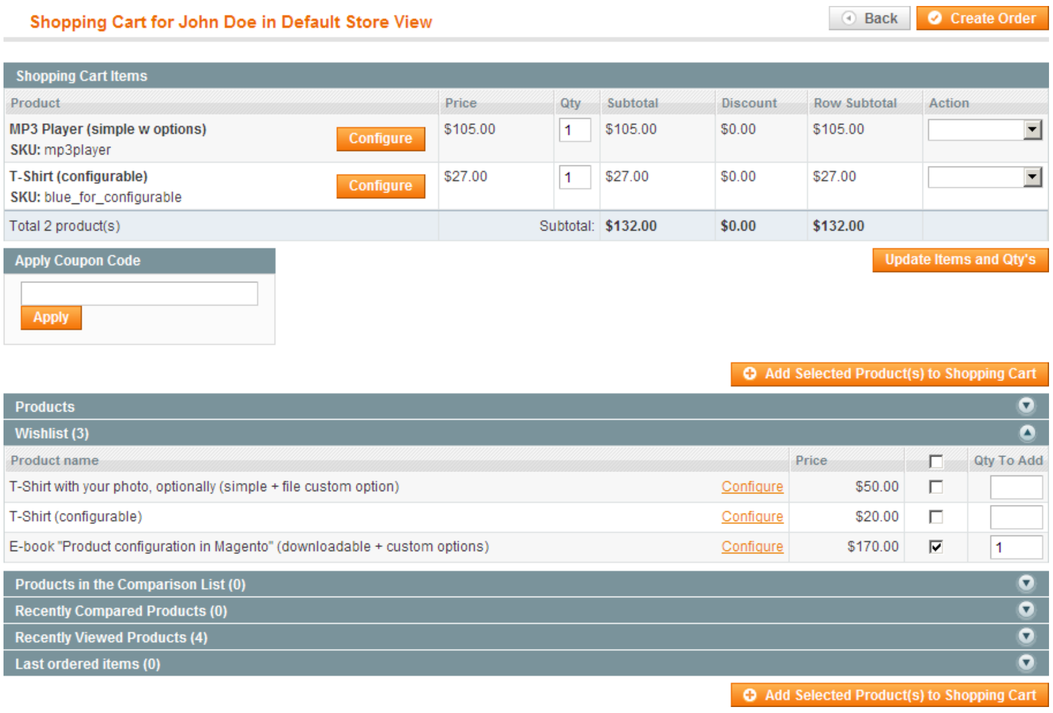
The Bundle product illustrates it nicely – look how it embeds the configuration helper into the Wishlist tab by using Dependency Injection via the public method `addProductConfigurationHelper()` in its layout file:

```
/app/design/adminhtml/default/default/layout/bundle.xml

<adminhtml_customer_wishlist>
    <reference name="customer.wishlist.edit.tab">
        <action method="addProductConfigurationHelper">
            <type>bundle</type>
            <name>
                bundle/catalog_product_configuration
            </name>
        </action>
    </reference>
</adminhtml_customer_wishlist>
```

Follow the same approach to add these instructions into your custom product type's layout file, and the store manager will see your product's options in the Wishlist tab together with built-in product types.

Manage Customer Shopping Cart Page (Magento EE)

Magento Enterprise Edition has an additional page (Fig. 60) – Manage Customer Shopping Cart – that provides advanced manipulation of a customer's Shopping Cart.



Shopping Cart for John Doe in Default Store View

Shopping Cart Items

Product		Price	Qty	Subtotal	Discount	Row Subtotal	Action
MP3 Player (simple w options) **SKU:** mp3player	Configure	$105.00	1	$105.00	$0.00	$105.00	
T-Shirt (configurable) **SKU:** blue_for_configurable	Configure	$27.00	1	$27.00	$0.00	$27.00	
Total 2 product(s)			Subtotal:	**$132.00**	**$0.00**	**$132.00**	

Apply Coupon Code

Products

Wishlist (3)

Product name		Price		Qty To Add
T-Shirt with your photo, optionally (simple + file custom option)	Configure	$50.00	☐	
T-Shirt (configurable)	Configure	$20.00	☐	
E-book "Product configuration in Magento" (downloadable + custom options)	Configure	$170.00	☑	1

Products in the Comparison List (0)

Recently Compared Products (0)

Recently Viewed Products (4)

Last ordered items (0)

Fig. 60. The Manage Customer Shopping Cart page in the Magento EE Admin Panel

Visually the page looks like a smaller copy of the Create Order page – here you can add, change, and remove items from the Shopping Cart. Internally it looks the same too – it's a mini-copy of the Create Order page's architecture.

You don't need to know anything more there. Keep this chapter as a reference and remember this page when developing any more configuration functions for Magento Enterprise users.

Magento Developer's Checklists

Congratulations! You have successfully read the whole guide for Composite Products and managing product configurations. Extremely useful, this guide will serve as helpful documentation for everyone who works with Magento.

To summarize the guide, I have prepared two checklists for Magento developers. They will help you remember the things that need to be implemented in order to support Composite Products and successfully manage product configurations in Magento. The checklists gather key points of information scattered throughout this guide. The first one is written for creators of custom product types, the second is for Magento theme builders.

Developing a Custom Product Type

All your own product types that provide configuration controls for a shopper fall under the Composite Products functionality. Remember to complete the following steps to provide your users with a solid Magento-style experience:

- Implement the `_prepareProduct()` method in a type model to parse each incoming POST-request into product options; options must implement the Configuration Item Option interface.
- Implement the `processBuyRequest()` method in a type model to transform a buyRequest into option values for the *Edit configuration* mode.
- Create frontend configuration controls for the View Product page that can display preconfigured values in *Edit configuration* mode.
- If you have file upload controls:
  1. their validation models should add the files to the file copy queue after successful validation.
  2. their rendering blocks should be able to work in the *Edit configuration* mode: show the user the current configuration and allow uploaded files to be changed/ deleted.
  3. you need to implement the ability to download the files in the Wishlist and the Shopping Cart controllers, the same way Custom Options files can be downloaded, when viewing an item's configuration.
- Create the Configuration Helper for your product type. It should return an array of option values specific to this product type, and merge them with the option values (custom options) that are inherited from basic product type.
- Register the Configuration Helper in the layouts to show the product options in the frontend Shopping Cart, Wishlist pages and the Customer Management pages in the Admin Panel (Wishlist, Shopping Cart and Manage Shopping Cart in Magento EE).
- Add your product type to the “available product types” so it will be used in the Create Order page in the Admin Panel.
- Create the backend configuration controls (inherited from frontend controls, but having the template and the Javascript suitable for the backend specifics).
- Adjust the `_canConfigure` property and/or the `canConfigure()` method in the product type model so the *Configure* buttons and links will appear in the Admin Panel when needed.

Creating a Store View Theme

The theme-builder's checklist is smaller. It features places where you should make your theme behave the same as the Core's built-in theme (if you have overwritten any parts of it):

- Whenever you overwrite layouts for the Wishlist, Checkout or Adminhtml modules, add the instructions for embedding Configuration Helpers – just copy these handles from the default Magento layouts.
- Use the Wishlist Item Collection instead of the Wishlist Product Collection in your blocks and templates. In your templates switch to using the Wishlist Item Collection's items instead of Wishlist Product Collection's items.
- Modify all configuration controls so they use preconfigured values in *Edit configuration* mode on the View Product page.
- Implement submitting products to the Wishlist through a POST-request (the whole form is submitted), not using the older GET-request (only product id is sent) on the View Product page.
- Change the Custom Option File template on the View Product page so this type of custom option is rendered differently in *Edit configuration* mode.
- Add the original Adminhtml layout handles that build the configuration popup to your custom Adminhtml layout file (if you have overwritten it).

Afterword

I'm really happy you went through the whole guide to Magento Composite Products functionality. Thanks for reading it!

The Magento Core team has spent a lot of time implementing and polishing the Composite Products functionality in release 1.5. With this guide, I hope these changes will help you, as a developer or store owner, to understand Magento better and achieve your goals in e-commerce quickly and easily.

Have fun working with Magento – simply the best e-commerce platform on the market!

Andrey Tserkus,
April, 2012