The supply chain of enterprise software
Postmus, D.

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2009

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):
Chapter 2

A typology of product delivery strategies in the software industry

2.1 Introduction

In the information systems literature, a distinction is often made between standard software and bespoke software (Sawyer, 2000; Carlshamre et al., 2001; Sommerville, 2004; Xu and Brinkkemper, 2007). Producers of standard software deliver high-volume products to many different customers. As a result, these products can generally be sold at just a fraction of their development costs. A disadvantage of a standard product is that its capabilities are determined and controlled by the producer of the software rather than the organization that is using the software. To account for the needs of different types of customer, producers of standard software often include parameters that enable a customer to tune the system by changing the corresponding parameter values, such as the preferred language. The use of such variation points can be further developed by the software vendor by allowing customers to add additional variants to the list of choices. Producers of bespoke software, in contrast, develop their products completely to customer specifications. These products therefore fit well with the customer’s existing business practices. The downside of a bespoke product is that it is generally sold at a price that is close to its development cost because it is difficult to sell this product to other customers.

Intermediate forms between standard and bespoke software exist as well. For example, when software product line engineering is applied, customers can configure their own software systems by selecting from a common base of core assets, which includes a generic architecture and a set of component implementations (Bosch, 2000; Clements and Northrop, 2002). Component-based software engineering is another approach where software systems are obtained that are partly standard and partly bespoke (Brown and Wallnau, 1998; Crnkovic et al., 2002). In both approaches, a compromise has been found between low average cost per delivered software system (such as with
standard software) on the one hand and the flexibility to meet customer-specific requirements (such as with bespoke software) on the other hand: there is some possibility to account for the needs of individual customers, while prices remain relatively low through the use of standardized components.

In the design and manufacturing of physical goods, forms in between mass production and craftsmanship have already been studied rigorously. An important concept from this area of research is the so-called customer-order decoupling point, which is defined as the point that “separates the part of the organization oriented towards customer orders from the part of the organization based on planning” (Hoekstra and Romme, 1992). The degree of customer specification freedom, i.e. “the extent to which the customer is allowed to deviate from the standards incorporated in the pre-defined product descriptions which were developed independently of the customer order” (Muntslag, 1993), is also important to consider because it determines the amount of freedom a customer is given to adjust an existing product to his or her specific requirements. When combined, the position of the customer-order decoupling point and the degree of customer specification freedom determine a vendor organization’s product delivery strategy.

By adapting the two concepts from the design and manufacturing of physical goods to the specific characteristics of the software industry, this chapter identifies two dimensions along which the product delivery strategies of software companies can be classified. The typology that results from combining these two dimensions contributes to a better and systematic understanding of the different positions software companies can take on the flexibility - cost spectrum. To put it another way, there is a trade-off between flexibility, i.e. the ability to produce and deliver customized products, and the average cost per delivered software system. When combined, the position of the customer-order decoupling point and the degree of customer specification freedom determine a software company’s relative position on the continuum ranging from low flexibility and cost (standard software) to high flexibility and cost (bespoke software). The typological issues discussed in this chapter may also be helpful to support software companies in selecting an appropriate product delivery strategy.

The remainder of this chapter is organized as follows. First, an overview is provided of relevant concepts from the field of operations management. Then, several important properties of the software development process are described. Subsequently, our framework for classifying the product delivery
strategies of software companies is derived by applying the two existing con-
cepts from the design and manufacturing of physical goods to the new field
of enterprise application software. Next, the notion of customization is ana-
yzed from a customer perspective by focusing on the risks the customer is
running when using the provided specification freedom to tailor the system
to his or her specific requirements. Finally, the chapter concludes with a brief
summary and suggestions for further research.

2.2 Theoretical framework

The production process of a manufacturing organization can be accurately
described by making a distinction between production phases and stock points.
A production phase is a process in which goods change intrinsically, whereas
a stock point indicates the storage of goods between two production phases
(Hoekstra and Romme, 1992). Stock in a stock point is waiting for a decision
with regard to further transformation. As a result, stock points add flexibility
to a manufacturing process: the products that are produced in the production
phase upstream of a stock point can be transformed into several different
products in the production phase downstream of the stock point.

Figure 2.1 shows how production phases and stock points can be used to
describe a production process. In the first production phase, raw materials
are transformed into subassemblies, which, in turn, are transformed into fin-
ished products in the second production phase. The stock point between the
fabrication and the assembly phase results in a decoupling of these phases:
if there are sufficient subassemblies in stock, the assembly phase can be per-
formed independently of the fabrication phase.

Figure 2.1: The manufacturing process of an organization. Rectangles reflect pro-
duction phases, and triangles reflect stock points.

A stock point of particular interest is the so-called customer-order decou-
pling point (CODP), also referred to as the order penetration point (Olhager,
2003) or the physical supply decoupling point (Wikner and Wong, 2007).
The CODP indicates at what stage in the production process forecast-driven
Chapter 2. A typology of product delivery strategies

and customer-order-driven production are separated (Hoekstra and Romme, 1992; Naylor et al., 1999; Van Donk, 2001). By varying the position of the CODP, three different levels of customer-order-driven production can be distinguished. These are:

- **Make-to-order (MTO)**. Production is completely customer-order-driven.
- **Assemble-to-order (ATO)**. Part of production is customer-order-driven and part forecast-driven.
- **Make-to-stock (MTS)**. Production is completely forecast-driven.

Wikner and Rudberg (2005) extend the classical CODP typology by adding an engineering dimension. This dimension of decoupling points indicates to what extent engineering has already been completed before a given customer order is accepted. The authors use the term *engineer-to-order (ETO)* to describe the situation when a product is designed to customer specifications, and they use the term *engineer-to-stock (ETS)* to describe the situation when a product is fully designed before the company gets an actual customer order. Finally, the term *adapt-to-order (ATO)* is used to cover positions of the CODP in between engineer-to-order and engineer-to-stock.

When combined, the two dimensions yield the classification framework depicted in Table 2.1. Following Wikner and Rudberg (2005), we use the subscripts ED for the engineering dimension and PD for the production dimension. A product must be fully engineered before it can be produced, so not every combination is feasible. For example, the material requirements and manufacturing instructions of an office chair must be available before the chair can be assembled. A plus-sign indicates a feasible combination; a minus-sign indicates an infeasible combination.

<table>
<thead>
<tr>
<th></th>
<th>MTO\textsubscript{PD}</th>
<th>ATO\textsubscript{PD}</th>
<th>MTS\textsubscript{PD}</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETO\textsubscript{ED}</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ATO\textsubscript{ED}</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>ETS\textsubscript{ED}</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

In addition to the position of the CODP, organizations may differ with respect to the degree to which customers are allowed to make changes to
product designs that have been obtained independently of any customer order. Muntslag (1993) distinguishes five levels of *customer specification freedom*, varying from interfacing the product with the customer’s current environment to modifying the performance level of the ultimate function of the product. Whereas the position of the CODP is related to a vendor’s investment in products prior to receiving a customer order, the degree of customer specification freedom indicates to what extent the customer is allowed to customize these products to his or her specific requirements. When combined, the position of the CODP and the degree of customer specification freedom determine an organization’s *product delivery strategy*, i.e. different product delivery strategies relate to different positions of the CODP and/or different degrees of customer specification freedom.

### 2.3 The software development process

Figure 2.2 shows that the development of software consists of four main activities: requirements specification, software design, implementation and testing, and deployment. During requirements specification, the functional and non-functional requirements for the software system are determined and documented. Software design, which includes both architectural design and detailed design, is concerned with creating a design that satisfies the system requirements. During implementation and testing, the design is transformed into a working software system that can be sold and delivered to the customer. Finally, during deployment, the system is brought into effective action by installing it on the customer’s hardware.

The waterfall model is the most straightforward model of the software development process (Messerschmitt and Szyperki, 2003; Sommerville, 2004). In this model, the process activities are performed sequentially: software design does not begin before requirements specification has been completed and implementation does not begin before the system has been designed. Many modern software development approaches, however, are not strictly sequential. Rather, they pass through the process activities in a series of iterations such that in each iteration part of the software system is produced (Kruchten, 2003).

Software reuse, i.e. “the process of creating new software systems from existing software rather than building it from scratch” (Krueger, 1992), is an important strategy for increasing the efficiency of the software development process. Software companies that apply software reuse systematically try to
advance the software development process by investing in the up-front development of one or more repositories of reusable software assets (Kim and Stohr, 1998). Analogue to stock points in the production of physical goods, libraries of reusable software assets add flexibility to the software development process: a reusable asset can be shared across several different software systems, provided that the asset has sufficient variation points (see Chapter 3 for a thorough discussion) to adjust it to the specific requirements of these systems. Systematic application of software reuse therefore allows software companies to divide their development processes into two or more separately controllable phases, which may again be broken down into several iterations. The production of reusable software assets, often referred to as development for reuse, takes place in the first development phase, whereas concrete software systems are produced in the final development phase, also referred to as development with reuse. As will be further elaborated on in Chapter 3, intermediate phases exist when software reuse is applied recursively, i.e. when software systems are obtained by reusing and extending one or more software assets that have also been developed with reuse.

### 2.4 A framework for classifying product delivery strategies

There is a trade-off between the average cost per delivered software system on the one hand and the flexibility to incorporate customer-specific require-
ments on the other hand. A software company’s position on the continuum ranging from low flexibility and cost to high flexibility and cost is determined by its product delivery strategy. More specifically, the term product delivery strategy is used to refer to a set of strategic choices that contribute to the achievement of a software company’s performance objectives. Here, we develop a generic framework for the classification of product delivery strategies based on two key decisions: the position of the CODP and the degree of customer specification freedom.

2.4.1 Position of the CODP

In this section, the engineering dimension of the CODP (CODP\textsubscript{ED}) is related to the specific characteristics of the software development process to arrive at three different degrees of customer-order-driven engineering in the software industry. Wikner and Rudberg (2005) introduce the term adapt-to-order to cover situations in between engineer-to-order and engineer-to-stock, where engineering modifications to existing product designs are accepted. We prefer to use our own term \textit{compose-to-order} to better reflect the actual engineering activities that are being performed at this position of the CODP\textsubscript{ED}.

In addition to engineering-related activities, software companies also perform production-related activities, such as compilation, build, product integration, and packaging and release of the product and its corresponding documentation. In industries based on physical goods, the position of the CODP\textsubscript{PD} is an important strategic decision because production is generally resource-intensive and the holding of stock is not without financial risks. For a producer of (enterprise) software, however, the situation is different: the marginal cost of producing one additional copy of the software system is negligible (Greenfield and Short, 2004; Xu and Brinkkemper, 2007). A software company’s relative performance on the competitive priorities of flexibility and cost is therefore primarily determined by the position of the CODP\textsubscript{ED}. For this reason, the positioning of the CODP\textsubscript{PD} is not considered in the discussion below.

\textbf{Engineer-to-order (ETO\textsubscript{ED})}

This position of the CODP\textsubscript{ED} corresponds to the case when the software development process is completely customer-order-driven. An important characteristic of the ETO\textsubscript{ED} position is therefore that the system requirements are
fully specified and controlled by the customer that commissions the software. Although software engineers may ‘copy-and-paste’ pieces of source code from existing systems to increase the efficiency of the software development process, software reuse is not explicitly planned for. This is in contrast to the next position of the CODP, where software reuse serves as a means for dividing the software development process into a part that is forecast-driven and a part that is customer-order-driven.

**Compose-to-order (CTO)**

This position of the CODP corresponds to the case when customer-specific systems are created by composing standardized components. Although the system requirements are still determined by the customer, they may be slightly adjusted to better fit the building blocks at hand. In many situations, the components being reused do not provide sufficient functionality to satisfy the requirements of a particular customer completely (even after modification). One or more custom-built components are then created to implement the ‘missing’ functionality.

An important characteristic of the CTO position is that the software development process is partly forecast-driven and partly customer-order-driven. Figure 2.3 illustrates this graphically. The collection of reusable components at the bottom of the figure has been developed independently of any customer order. During the customer-order-driven part of the software development process, three of these components (A1, A3, and A5) are composed with two custom-built components (B1 and B2) to create the system that is depicted at the top of the figure. The integration code that is required to connect the five components is shown in white color.

A real-life example of CTO is provided by Sherif and Vinze (2002), who describe how an oil and gas company has successfully created a repository of reusable components for the sub-surfacing domain. The authors argue that within the oil and gas company, “the compelling need for a software repository originated from the fact that several disciplines in the sub-surfacing domain, like exploration and production, reservoir engineering, seismic interpretation, geophysics, geology, and many others, had discovered they were dissipating their resources by repetitively solving similar problems”. To overcome this inefficiency, the company decided to set up a working group that was given the responsibility of designing and implementing a repository of reusable assets to be shared among the application groups of the various dis-
A framework for classifying product delivery strategies

Figure 2.3: Creating a customer-specific system (top) by composing standardized components (bottom).

ciplines. Although it took the working group several years to populate the repository, the reuse initiative is considered to be a success as the application groups are now able to deliver their systems in significantly less time and with fewer resources.

Engineer-to-stock (ETS_{ED})

This position of the CODP_{ED} corresponds to the case when the software development process is completely forecast-driven. An important characteristic of the ETS_{ED} position is therefore that the system requirements are fully controlled by the organization that develops the software.

For each software system that is produced on forecast, a decision has to be made with regard to the number of variants a customer can choose from. A variant is an instance of a software system that differs, in some way, from other instances (Sommerville, 2004). For example, variants may differ with respect to the functionality provided or may have been designed to work with different operating systems. Lampel and Mintzberg (1996) identify three distinct strategies: pure standardization, segmented standardization, and customized standardization.

In the case of pure standardization, all customers receive the same variant of the software system. This strategy is usually adopted by producers of professional application software, such as computer-aided design (CAD) and simulation software. A software vendor may also decide to release two or more system variants. This strategy is known as segmented standardization.
and is applied by companies that sell their products to different types of customer, such as consumers, SMEs, and companies with complex IT infrastructures. Microsoft, for example has released six variants of its Windows Vista operating system, each targeted at a different market segment. Finally, when customized standardization is adopted, the customer is allowed to specify his or her own variant of the system. This strategy is often applied by producers of off-the-shelf enterprise software, such as SAP or Oracle. An enterprise application suite such as SAP Business Suite, for example, consists of several integrated applications, such as an enterprise resource planning (ERP) application, a customer relationship management (CRM) application, and a business intelligence (BI) application. For each of these applications, one or more alternative implementations exist, which may (or may not) be selected by the customer.

2.4.2 Degree of customer specification freedom

The position of the CODP\textsubscript{ED} discussed above determines to what extent a software company caters for the needs of individual customers by producing and delivering custom-built systems. However, the vendor company may also decide to postpone product differentiation by allowing customers to customize the system after it has been delivered (i.e. during deployment), such as adding a new interface or changing the order in which certain activities are executed. The degree of customer specification freedom indicates to what extent the customer (or the vendor company / an external organization, such as a systems integrator or reseller, on commission of the customer) is enabled to perform such post-delivery changes.

The term customer specification freedom is introduced by Muntslag (1993), who uses it to classify producers of industrial machinery into different categories based on the extent to which the customer is allowed to make changes to existing product designs in view of the engineer-to-order character of these kind of products. The author perceives customer specification freedom as a one-dimensional concept in the sense that he identifies five different levels (ranging from interfacing the product with the customer’s current environment to modifying the performance level of the ultimate function of the product), which from a vendor’s perspective can be ordered according to an increasing technical risk. Just like producers of industrial machinery, software companies may support various types of changes. In contrast to Muntslag (1993), however, we do not try to order these changes based on some kind
2.4. A framework for classifying product delivery strategies

of criterion, such as the amount of technical risk. Instead, we consider customer specification freedom to be a multi-dimensional construct, where each dimension relates to a different type of customization, which may or may not be supported by the software vendor. In particular, the following three dimensions of customer specification freedom are distinguished (see Figure 2.4):

1. accessing the system’s functionality through an application programming interface or web service,

2. configuring the behavior of the software system (either by setting parameters or through the use of extension points),

3. adapting the behavior of the system.

![Figure 2.4: Three dimensions of customer specification freedom.](image)

**Accessing the system’s functionality through an application programming interface or web service**

The first dimension of customer specification freedom is concerned with accessing the system’s functionality through an application programming interface (API). An API of a software system consists of an operation that can
be invoked by other, external systems to perform a certain task, such as retrieving information from the system’s database (e.g. the status of a customer order). The software system may also support operations for which no API is defined, but those cannot be called by other systems. Figure 2.5 illustrates this point graphically. The external system at the left of the figure can access part of the system’s functionality by invoking the operations that are labeled as operation A() and operation B(). For the operation that is labeled as operation C(), however, no API is declared, and this operation can therefore only be invoked from within the system.

![Diagram](image)

**Figure 2.5:** Accessing the system’s functionality through its application programming interface.

APIs serve as a mechanism for achieving interoperability between software systems. For example, a producer of consumer goods may want to integrate a newly obtained e-commerce system with its existing inventory management system to provide its online customers with reliable predictions on the expected delivery date of the selected product (e.g. by checking whether this product is in stock). The company will not be able to connect the two systems if the inventory management system does not have an API through which the e-commerce system can retrieve this information.

Nowadays, the API will often take the form of a web service, thereby leveraging standards and technologies of the Internet. A software product which is intended to work in a stand-alone manner, without integration to other software, will not provide APIs or web services. Such products are often consumer software (such as computer games), but these can also be found as business applications (e.g. specialized professional applications such as
2.4. A framework for classifying product delivery strategies

simulation software).

**Configuring the behavior of the software system**

The second type of customer specification freedom is concerned with configuring the behavior of the software system to meet the specific needs of the customer. The predefined places at which this behavior can be adjusted are generally referred to as variation points. Each variation point is associated with a number of alternative choices called variants. A variation point is bound by selecting a certain variant from the set of associated variants.

Two mechanisms for realizing this type of customer specification freedom can be distinguished: parameterization and extension points (Bosch, 2000). Parameterization refers to the case when variation points are implemented as switches in the system’s source code. To assist the customer in fine-tuning the system, a configuration interface is provided that allows him or her to modify the behavior at these switches by changing the corresponding parameter settings. When extension points are used, the software system is divided into two parts: a part that contains stable functionality and a part that contains functionality that may vary across customers. The variable part of the system is modeled as a set of independent entities (extensions) that can be plugged into the stable part of the system to specialize its behavior. The different instances of these extensions are implemented as variants and the customer can configure the system by either selecting an existing variant or developing a new variant (Bosch, 2000). To ensure interoperability between the stable and variable part of the system, a so-called service provider interface is provided to which all extensions must conform (Seacord and Wrage, 2002).

A mature software system nearly always contains parameters to enable a customer to adjust the system to his or her specific situation. An enterprise resource planning (ERP) system, for example, contains thousands of parameters that allow the customer to configure the system by choosing between different executions of its functions and processes (Brehm et al., 2001). Moreover, if the variety of customer wishes for a particular piece of functionality becomes very large, the possibility of implementing this functionality as an extension point can be considered.

**Adapting the behavior of the system**

The third dimension of customer specification freedom is concerned with adapting the behavior of the system. While configuration is a form of black-
box customization (i.e. a customer only needs to understand the system’s configuration and/or service provider interfaces), adaptation requires the customer to have a detailed understanding of the internal structure of the system. Software vendors that allow customers to adapt the system therefore operate on markets where the customer has a professional IT staff that can be contracted to take responsibility for the implemented changes.

Inheritance is a well-known technique of white-box adaptation. It allows the customer to change the behavior of the system by overriding its existing methods and can be applied when the system is implemented in an object-oriented programming language, such as Java or C#. In practice, inheritance is often used in the context of an application framework. Application frameworks are semi-finished systems consisting of a collection of concrete and abstract classes that can be tailored to specialized implementations by subclassing (Fayad and Schmidt, 1997). The abstract classes specify the flow of execution and define certain methods called hot spots (Pree, 1994) that can be overwritten by the customer. The concrete classes provide default implementations of the framework’s abstract classes and can usually not be adapted (Bäumer et al., 1997).

Modifying the system by rewriting part of its source code is another technique through which adaptation can be achieved. The making of source code changes is often strongly discouraged since it involves high risks. For example, Stedman (2000) reports that Volkswagen was experiencing significant problems in delivering spare parts to its dealers after it had decided to modify its ERP system (SAP R/3 in this case). Because of these risks, some software vendors explicitly prohibit the making of source code changes. Others are willing to give their customers access to the source code, but may refuse to provide future support for the system after code changes have been made (Brehm et al., 2001).

2.4.3 A typology of product delivery strategies

When combined, the position of the CODP_{Ed} and the degree of customer specification freedom yield the typology of product delivery strategies as depicted in Figure 2.6. Our classification scheme consists of three generic product delivery strategies, each corresponding to a different position of the CODP_{Ed}. The degree of customer specification freedom is used as a secondary characteristic to divide these three generic strategies into zero or more subtypes, depending on the kind of customizations the customer is allowed to
2.4. A framework for classifying product delivery strategies

make:

- When the CODP\textsubscript{ED} is located at the ETS\textsubscript{ED} position, one or more off-the-shelf software systems are delivered to customers. The degree of customer specification freedom can then be used as a secondary characteristic to split this generic strategy into eight distinct subtypes, based on different scores on the three dimensions discussed above.

- The case when the CODP\textsubscript{ED} is located at the CTO\textsubscript{ED} position is slightly different. Now, the concept of customer specification freedom can be applied at the component level to indicate to what extent the software vendor’s application engineering department is allowed to tailor the reusable components (produced by the domain engineering department) to the situation at hand.

- Finally, when the CODP\textsubscript{ED} is located at the ETO\textsubscript{ED} position, the degree of customer specification freedom cannot be seen as a strategic decision to be made by the software vendor because there are no software assets produced on forecast.

![Diagram of product delivery strategies](image)

**Figure 2.6:** A typology of product delivery strategies in the software industry.

Tables 2.2 and 2.3 serve to illustrate our typology by applying it to some examples. The off-the-shelf software systems listed in Table 2.2 differ with respect to the degree of specification freedom that is given to the customer. Each of these examples therefore corresponds to a different subtype of the ETS\textsubscript{ED} strategy. Software companies that are specialized in producing computer games, for example, are characterized by a relatively low degree of customer specification freedom: they allow for configuration in the form
of parameterization (e.g. for setting general preferences, such as language, screen resolution, and sound level), but do usually not support any of the other forms of customization. Producers of ERP software, on the other hand, are characterized by a relatively high degree of customer specification freedom: besides extensive parameterization, ERP systems can be integrated with other systems, extended by adding additional business processes and menu structures, and sometimes even be adapted by adding new database fields and tables and/or rewriting the system’s source code (Brehm et al., 2001; Ahituv et al., 2002).

The examples listed in Table 2.3 correspond to different subtypes of the CTOED strategy. A software component always have an API through which other components can access its functionality. Not every position in the three-dimensional space that results from combining the dimensions of customer specification freedom therefore applies to this position of the CODPED.

2.5 Risks involved in making changes to a software system

The previous section discussed customization from a vendor perspective by focusing on the amount of freedom a customer is given to introduce changes and enhancement to a system after it has been delivered. In this section, the concept of customer specification freedom is analyzed from a customer perspective by focusing on the risks the customer is running when adjusting the system to his or her specific requirements.

From a customer perspective, there are two risks involved in making changes to a software system: (i) the risk of jeopardizing the vendor’s warranty and support for the system, and (ii) the risk of jeopardizing seamless upgrading of the system. Based on these two risks, the four different types of customization identified and discussed in the previous section (i.e. APIs, parameters, extension points, and adaptation) can be classified into three categories: changes that jeopardize neither the vendor’s warranty and support nor seamless upgrading of the system, changes that jeopardize seamless upgrading of the system but do not result in loss of vendor support, and changes that jeopardize both the vendor’s warranty and support as well as seamless upgrading of the system (see Figure 2.7). In the previous section, it was already mentioned that adaptation both endangers the vendor’s warranty for proper functioning of the system and hinders smooth upgrading to major
Table 2.2: Examples of different subtypes of ETS\textsubscript{ED}.

<table>
<thead>
<tr>
<th>API</th>
<th>Parameters</th>
<th>Extension points</th>
<th>Adaptation</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Consumer applications (e.g. computer games)</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Stand-alone professional applications (e.g. CAD software)</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Application frameworks for stand-alone systems</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Office applications (e.g. word processors)</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Low-end enterprise applications (e.g. ERP systems targeted at SMEs)</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>High-end enterprise applications (e.g. ERP systems targeted at large companies with complex IT infrastructures)</td>
</tr>
</tbody>
</table>
Table 2.3: Examples of different subtypes of $CTO_{ED}$.

<table>
<thead>
<tr>
<th>API</th>
<th>Parameters</th>
<th>Extension points</th>
<th>Adaptation</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Traditional components</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>White-box frameworks as components</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Black-box frameworks as components</td>
</tr>
</tbody>
</table>

new versions. Adaptation is therefore positioned in the upper left corner of Figure 2.7. The positioning of the other three types of customization in their appropriate cell(s) is briefly discussed in the paragraphs below.

APIs are provided by a software vendor to enable customers to integrate the system with other, external systems. The use of these interfaces should therefore not jeopardize the vendor’s warranty and support for the current version of the system. The situation is slightly different when upgrades are considered. Although most software vendors will strive for continued support of existing APIs or web services, customers always run the risk that some of these services are no longer available in future releases. Even if an upgrade continues all APIs used by a particular customer, the customer still has to test the existing connections with the new version of the system.

Parameters and extension points are used by a software vendor with the purpose of providing variation points. The use of these variation points should therefore not result in loss of vendor support for the system. However, only parameterization can be fully foreseen and tested by the software vendor. When extensions are considered, a distinction can be made between extensions that leave the system’s database untouched and extensions that result in additional database fields and tables. This latter type of extension generally leads to extra sessions and business logic and may affect database integrity.

As far as upgrades are concerned, parameters and extension points will generally remain stable over service packs (small upgrades, which do not affect the structure of the database and offered functionality), but cannot be assumed to remain present over new releases (larger upgrades, which enhance the system’s existing functionality). Extensions that result in additional
database fields and tables hinder smooth upgrading to major new releases because such upgrades usually require database changes.

<table>
<thead>
<tr>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>* Adaptation</td>
<td>* Extension points</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>* APIs or web services</td>
<td>* Parameters</td>
</tr>
<tr>
<td>* Extension points</td>
<td></td>
</tr>
</tbody>
</table>

Jeopardize the vendor’s warranty and support

Figure 2.7: Risks involved in making changes to a software system.

2.6 Conclusion

To contribute to a better and systematic understanding of the different positions software companies can take on the continuum ranging from low flexibility and cost (standard software) to high flexibility and cost (bespoke software), this chapter applies two existing concepts from the design and manufacturing of physical goods to the new field of enterprise application software: the customer-order decoupling point (particularly its engineering design perspective) and the degree of customer specification freedom. The position of the customer-order decoupling point determines to what extent a software vendor caters for the needs of individual customers by producing and delivering custom-built systems. It is used as a primary characteristic to divide the product delivery strategies of software companies into three generic categories: engineer-to-order, compose-to-order, and engineer-to-stock. The degree of customer specification freedom is used as a secondary characteristic to split these three categories into several subtypes, depending on what kind of customizations a customer is allowed to make.

Apart from looking at it from a vendor’s perspective, the notion of customization can also be analyzed from the viewpoint of a customer, by focus-
ing on the amount of risk the customer is running when using the provided specification freedom to tailor the system to his or her specific requirements. Generally speaking, there are two risks involved in making changes to a software system: the risk of jeopardizing the vendor’s warranty and support for the system, and the risk of jeopardizing seamless upgrading of the system. Based on these two risks, the customizations made by a customer can be categorized into three groups: changes that jeopardize neither the vendor’s warranty and support nor seamless upgrading of the system, changes that jeopardize seamless upgrading of the system but do not result in loss of vendor support, and changes that jeopardize both the vendor’s warranty and support as well as seamless upgrading of the system.

Future research may be directed at extending our typology by identifying additional dimensions along which the product delivery strategies of software companies can be described and classified. It may also be worthwhile to analyze the relationship between a software vendor’s product delivery strategy and its performance on the competitive priorities of flexibility and cost efficiency in more detail, so that normative rules for selecting an appropriate product delivery strategy can be derived. Finally, it would be interesting to empirically validate and test our classification framework by applying it to several case studies.