Introducing Software Reuse 1

ABSTRACT

This opening Chapter offers an overview of the main issues in software reuse. It defines what we mean by reuse, and discusses a number of fundamental concepts essential to a balanced understanding of reuse.

1.1 First of All . . .

Of his many accomplishments, Erwin Schumacher is best remembered for telling us that 'small is beautiful'. His phrase became established in the late twentieth century mindset.

Software reuse, the subject of this book, is about a twofold promise, that software projects can be small, and that they can create beautiful software – that is if you accept that high quality, in the form of clean design, fitness for purpose and a low defect count, constitutes 'beauty' in software.

We have become accustomed to software projects that are ugly and uncontrollable monsters, devouring seemingly endless resources, and delivering products that are ill-structured, over-sized and bug-infested, and which often fail to meet our real needs. Under the name of 'the software crisis' we have accepted that state of things for thirty years. Fred Brooks' familiar and well-argued assertion that there is 'no silver bullet' may paradoxically have made things worse: convince people that there is no one simple medicine, and they may shrink from undergoing more radical treatments.

Properly understood, and deployed in the right context, reuse offers the opportunity to achieve radical improvement. It should still, however, not be regarded as a silver bullet, a simple recipe that we can depend on to rid us of monstrous projects and bad software.

In one sense this is a modest book. It seeks to present just the essentials of software reuse. It is neither an academic textbook, nor a cookbook with ready-made recipes telling practitioners or managers 'how to do it'. It offers, simply and without unnecessary detail or jargon, an introductory overview of the issues involved in the successful practice of reuse.

In another sense, however, the book's aim is far from modest. It is to set out the vision and concepts of reuse, and the experiences of some of its pioneers, in the hope that you may be encouraged to think about adopting reuse yourself as a new way of life. It aims, in other words, to be a gateway to the wonderland of reuse. If it succeeds in meeting that aim for you, so that you pass through the gateway, then will be the time to start exploring the large (and rapidly growing) body of excellent detailed literature on reuse, for more detailed information on cost models, class libraries, organizational structures, repository management, reuse maturity assessments, framework and component technologies and many other important issues. For now, we hope you will find in these pages excitement, surprise, challenge and new visions of what it is possible to achieve in the difficult business of creating software.

1.2 Definition and Basic Essentials

The objective of this introductory Chapter is to present an outline map of the reuse landscape, identifying the essential ideas of software reuse, before they are developed further in later Chapters. We start with a definition.

Definition

Software reuse is the systematic practice of developing software from a stock of building blocks, so that similarities in requirements and/or architecture between applications can be exploited to achieve substantial benefits in productivity, quality and business performance.

A definition cannot include everything that might be said about the term being defined. It should incorporate a choice of features that are necessary and sufficient for understanding the term. The above definition is based on four key features of software reuse.

- Reuse is a systematic software development practice.
- Reuse employs a stock of building blocks.
- Reuse exploits similarities in requirements and/or architecture between applications.
- Reuse offers substantial benefits in productivity, quality and business performance.

Let's now look at each of those ideas in turn.

1.2.1 Reuse Is a Systematic Software Development Practice

Software reuse has a wide spectrum of possible meanings. On the one hand, it is possible to interpret much of the progress in software practice in the past half-century in terms of increasing levels of reuse (see Section 1.4, later in this Chapter). On the other hand, advanced and sophisticated approaches to reuse exist at the current leading edge between research and practice, which are probably outside the competence range of most software developing organizations (so-called *generative reuse* is an example). The definition does not embrace that whole range of meanings, but focuses attention on the centre of the range – an approach to reuse which is practically feasible for most developers here and now, which is a substantial advance on accepted practice, and which can offer major benefits.

One result of that definition strategy is the inclusion of the word 'systematic'. 'What about non-systematic reuse', you may have wondered; 'isn't that still reuse?' You are right: of course it is! Strictly we have offended against logic in defining reuse as only a part of itself. It's like equating soccer with just the premier league game, or describing drama just in terms of Molière, Pirandello and Shakespeare.

The justification for offending against logic in that way is that it is not possible, except by undertaking software reuse *systematically*, to realize the bold claims made on its behalf. It is the bold claims, the large benefits, and the substantial commitment needed to achieve them, that form the subject-matter of this book. We do not want to waste your time telling you about

something that is everyday or casual, but rather to set out a big vision of how software capability can be transformed. Having said that, what distinguishes systematic from non-systematic reuse?

Some of the key features of the systematic practice of software reuse are set out in *Fig. 1.1*. Non-systematic reuse is, by contrast, ad hoc, dependent on individual knowledge and initiative, not deployed consistently throughout the organization, and subject to little if any management planning and control. If the parent software organisation is reasonably mature and well managed, it is not impossible for non-systematic reuse to achieve some good results. The more probable outcome, however, is that non-systematic reuse is chaotic in its effects, feeds the high-risk culture of individual heroics and fire-fighting, and amplifies problems and defects rather than damping them.

SYSTEMATIC SOFTWARE REUSE MEANS . . .

Understanding how reuse can contribute toward the goals of the whole business.

Defining a technical and management strategy to achieve maximum value from reuse.

Integrating reuse into the total software process, and into the software process improvement programme.

Ensuring all software staff have the necessary competence and motivation.

Establishing appropriate organisational, technical and budgetary support.

Using appropriate measurements to control reuse performance.

Fig. 1.1 Systematic software reuse.

1.2.2 Reuse Employs a Stock of Building Blocks

The term 'building block' is used in the definition because the properties of physical building blocks, and the way we use them, are well understood, and they convey very well the conceptual flavour that is appropriate for explaining software reuse. Let us consider some of those properties (*Fig. 1.2*).

Building blocks are not only children's playthings. The concept is widespread throughout industry, and it is intrinsic to product breakdown structures. The design of a subsystem, assembly, sub-assembly or part may be common to several different models of motor car, washing machine, industrial pump or machine tool. Even the complete architecture of a product may be carried over from one model to subsequent models. Reusable building blocks are fundamental to the goal of not reinventing the wheel, and to

Building blocks are artefacts, which can be put together to make larger-scale artefacts.

They may or may not have been designed primarily for use as building blocks (compare a box of toy blocks with off-cut pieces of wood).

They may or may not have been designed to fit together in a standard way (compare a box of toy blocks with the Lego system).

The greater the diversity of building blocks (in terms, for instance, of size, shape, colour, material), the greater the diversity of structures that can be created from them.

If a building block has unusual properties, its use is likely to be restricted to a smaller number of specific structures. The more general its properties, the more general its use.

There may be large building blocks that define a partial 'architecture' into which smaller ones can be fitted (for example, a Lego baseboard or chassis).

Building blocks may be combined into sub-assemblies, so that a sub-assembly can then be incorporated into various different end-products.

Fig. 1.2 Some important properties of building blocks.

making progress by building on previous experience of what works. It has been central to industrial development for the past two centuries.

Software reuse is based on exactly the same concepts as children's building blocks or reusable designs in industry. Although systematic reuse is not yet widely practised in software development, it will be seen in the future to be as essential to progress in software as it has been to progress in manufacturing and other sectors of industry.

In the software reuse literature, building blocks are most commonly called *reusable assets* (or *reuse assets*; often just *assets* for short), and we will use those terms from now on. Assets are work products of any kind, from any part of the software process. 'Asset' is an appropriate word, since software work products capture knowledge that is important to the enterprise, and therefore carry potential value. Reuse is a powerful means of exploiting that value-adding potential.

Assets may be of a technical or management nature, large-grained or fine-grained, simple or composite. They may have varying degrees of *leverage* (leverage is said to occur when reuse of one asset makes possible the reuse of a chain of other related assets further 'downstream' in the process). The belief by some people that source code modules are the only kind of reusable assets is mistaken; assets may include such things as requirements, project plans, estimates, architectures, designs, user interfaces, test plans, test cases, data, quality plans, and documentation. The concept of reusable assets is developed more fully in Chapter 2.

The definition refers to 'a *stock* of building blocks'. If that stock comprises more than a small number of items, we need to make arrangements to know what items we have, where to find them and whether they are worth keeping; otherwise we will not derive the best return from having established the stock. In this respect, a stock of reusable software assets is no different from any other kind of stock. A small shopkeeper can see or remember the contents of his store-room; the big store needs techniques of stock control and management. Managing a stock of reusable software assets, described in Chapter 3, uses concepts such as the asset *catalogue* and the asset *repository*.

Having an effective catalogue is often essential in achieving systematic reuse. Its absence can be a major contributor to chaos. It is rather like the situation, familiar no doubt to many parents, where Lego blocks are scattered all over the house, in every room, so that you don't know which ones you've got, the ones you need you can't find, and the ones you find by chance you probably don't need.

If a catalogue lists what assets we have and where they are stored, then a repository (like a library or a warehouse) is where we store and retrieve them. Reusable software assets may be stored in a single repository, or there may be multiple repositories: that is like the difference between keeping all Lego blocks in a single big box, or keeping the members of different sets in their separate boxes.

As can be easily imagined, how to design repositories for reusable software is of great interest to those of a technical inclination, and there is much discussion of it in the technical literature. Should we use database technology, or a configuration management system, or the repository of an integrated software engineering environment? Should the catalogue and repository functions be combined or separate? How should assets be classified for storage and retrieval, and how should search requirements be specified?

As already indicated, such questions may not be important to small software organisations (like small shopkeepers), whose asset collection is not big or complex enough to justify such a volume of technical fire-power. In this book the intention is to limit the discussion of catalogue and repository questions to the essential issues, and not to become entangled with too many technicalities.

1.2.3 Reuse Exploits Similarities in Requirements and/or Architecture Between Applications

This key idea is discussed in two stages. First, we will discuss what is meant by an application, and some related concepts. Second, we will look at how the potential for reuse arises from fundamental similarities between applications.

What Is Meant by Applications?

For our purposes, an *application* means a collection of one or more programs, together with all necessary supporting work products, that undertake some substantial user function. It may be alternatively called a system or a product in some contexts. Examples of applications might include word processing, payroll, engine control, seat reservations, vehicle routing, survey analysis and so on.

Note that the term *application domain*, which is an important concept in software reuse, is not the same as an individual *application*. The application domain of word processing, for example, refers to the general problem area of word processing. By contrast, an individual word processing application represents one specific solution within that general problem area. The same distinction applies in the case of payroll, engine control, and the other applications listed as examples in the previous paragraph.

An enterprise may develop one or more applications in the same domain. The need for different applications may arise because of the existence of a range of users with different needs, and/or because of changes in user needs over time.

The possibility of employing reuse to handle changes over time raises the question of whether maintenance is a form of reuse. Opinions differ. We would propose that maintenance and reuse are in principle independent, but in practice may be related. Maintenance, like initial development, can (but need not) employ reusable assets; reusable assets, like all software, need to be maintained.

There is no universal or clear-cut way of defining the boundaries between one application and another within a domain, or between one domain and another. Different organizations define such boundaries according to their own perceptions and business circumstances. In some cases organizations refer to applications in a common domain as a *product line*. Software product lines are discussed briefly later in this Chapter.

What Is the Source of Similarities Between Applications?

Why is it that a piece of source code, for example, or any other kind of asset, might prove to be usable in the context of two or more different applications, in the same domain or even perhaps in different domains? The reason must obviously be that there is a demand that the applications have in common, that can be satisfied by using common assets. The applications are *similar* in respect of that common demand.

A good illustration is provided by Reed (1995).[1] 'On Wall Street, some companies are structured as several small development teams, each supporting a trader. Most, if not all, of these teams are building similar applications. This type of development organization can capitalize substantially on software reuse.'

Of course, if two applications were similar in every respect, they would be identical. In that case, there would be no need to design and build the application a second time: the original application would simply be replicated. (Mere replication of a complete application is not regarded as reuse, according to the conventional usage of the term – just as we do not conventionally say that every Fiat Uno car is a *reuse* of the Fiat Uno design.) Thus, reuse of assets between one application and another is dependent on there being *both similarities and differences* between the applications – just as there are similarities and differences between the Uno and other Fiat models. The more similarities exist, the more potential there is for assets to be reused.

There are two ultimate sources of similarities among applications – their *requirements* and their *architecture*. Requirements specify the *problem*, that is the objectives of an application, in terms of a set of characteristics with which the eventual specific solution must comply. An architecture specifies what might be called a general category or mode of *solution*. Both requirements and architecture thus constrain or delimit the solution space within which the specific solution must be located, and it is those constraints or delimitations that are the root causes of similarities and differences between applications.

The process of determining requirements begins at the start of a project, and may continue, at greater or lesser intensity, until close to the end. Early attention focuses on high-level (or whole-system, or user) requirements. Those early requirements lead to design decisions that in turn generate further requirements at lower levels, perhaps for individual subsystems or parts.

Requirements may address any of the dimensions of the solution space, which include functionality, performance, reliability, maintainability, user

interface, cost, delivery date, operational platform (hardware and software), development platform (methods, languages and tools), portability, length of life, interactions with other systems (support systems, peer systems and contextual systems) and so on. Adopting a specified architecture may itself, in some circumstances, be a high-level requirement; conversely, it may be a design decision arising from the need to meet other requirements. In either case, adopting a specific architecture is likely to influence downstream requirements. In other words, requirements and architecture may not be wholly independent of each other.

Architectures are commonly thought of as defining categories or modes of solutions in three of the above dimensions: functionality, user interface, and operational platform. *Functional architectures* relate to application domains, and are consequently often referred to as domain architectures. Domain architectures are very important in reuse, and a good deal will be said about them later. *User interface architectures* define stylistic features of interfaces, such as menus, forms, command lines, hot buttons and so on, and the ways in which they are combined. *Operational platform architectures* (often called implementation architectures) define common patterns in which hardware, operating systems, other software utilities and middleware are configured to provide the infrastructural support which the application-specific software uses. User interface and operational platform architectures relate to what may be called technical domains.

In general, the term *vertical reuse* is used to refer to reuse which exploits functional similarities in a single application domain. It is contrasted with *horizontal reuse*, which exploits similarities across two or more application domains. There are two forms of horizontal reuse. The first refers to the exploitation of functional similarities across different domains; an example might be loans and reservations functions in the domains of libraries and car hire. The second refers to the exploitation of similarities in technical domains (user interface and operational platform), which are independent of application domains. This distinction is discussed further in Chapter 2.

Opportunities for reuse are like defects: the sooner they are found the better. If a defect is introduced during requirements definition, and is found at the same stage, the cost of correction is small; if it is not found until, say, a late stage of testing, the cost can be enormous. Likewise, if similarities between applications are identified at the stage when requirements and architecture are being determined, and if that leads at once to a recognition of opportunities for reuse, the potential benefit is hugely greater than if the opportunity is only recognized at, say, the coding stage.

Note that we say 'potential benefit'. Whether the benefit can be fully realized doesn't depend only on early recognition of similarities and of the

corresponding reuse opportunities. It also depends on how well the 'flow-down' from requirements and architectural decisions to later downstream decisions was recorded in the earlier system. If it was well recorded, then the chances of 'replaying' that flowdown in the new application are high, which means that a greater number of downstream assets can be reused and high leverage can be achieved.

We are now encountering again the difference between systematic and non-systematic reuse. Non-systematic reuse depends upon individuals casually recognizing similarities at any stage of development, based on their previous experience with other applications. It depends on their using their own initiative about whether to exploit those similarities, and whether to pass the information on to colleagues (who might or might not make use of it). Systematic reuse means a continual conscious and organized search for reuse opportunities in all work products, so that the opportunities are detected as early as possible, and so that maximum leverage is obtained. It also means augmenting the chances that future applications can exploit reuse opportunities, by building in traceability from upstream requirements and architecture decisions to other downstream work products.

Let us try to summarize. Opportunities for reuse from one application to another originate in their having similar requirements, or similar architectures, or both. The search for similarities should begin as close as possible to those points of origin – that is, when requirements are identified and architecture decisions are made. The possibilities for exploiting similarities should be maximized by having a development process that is designed and managed so as to give full visibility to the flowdown from requirements and architecture to all subsequent work products.

1.2.4 Reuse Offers Substantial Benefits in Productivity, Quality and Business Performance

Perhaps you are beginning to think that reuse seems a lot of effort. Right! In that case, you may also be wondering whether it is worth it. That is the question to which we now turn.

In orderly systems, you don't get something for nothing, and big changes usually demand big and sustained efforts. It is only by resting content in the arms of chaos that you open yourself to the chance of a small cause leading to a large change. Unfortunately, in a state of chaos, any change is unpredictable, and its effects are as likely to be harmful as they are to be beneficial. Prominent management writers have some striking things to say about achieving big changes for the better: see *Fig. 1.3*.

What companies require is seldom anything so 'reasonable' and 'realistic' as a 10 per cent improvement in some performance measure or other. What we actually require is more often something like a 50 per cent improvement, or a 75 per cent improvement.

(Michael Hammer/James Champy, talking about business process reengineering)

Good things happen only when planned; bad things happen on their own . . . One doesn't just go from awful to wonderful in a single bound.

(Philip Crosby, talking about quality)

Tackling a difficult problem is often a matter of seeing where the high leverage lies, a change which – with a minimum of effort – would lead to lasting significant improvement. High-leverage changes are usually non-obvious. So people shift the burden of the problem to other solutions – well intentioned, easy fixes, which seem extremely efficient. Unfortunately the easier solutions only treat the symptoms; they leave the underlying problem unaltered . . . and the system loses whatever abilities it had to solve the underlying problem.

(Peter Senge, talking about learning organizations)

Fig. 1.3 Some views on business change.

What these writers agree in saying, in different ways and from different viewpoints, is (a) big changes are both necessary and possible, (b) the changes we undertake should address business-critical problems and be based on understanding the root causes of those problems, (c) the changes will take a lot of sustained effort, but (d) the resulting value can far outweigh the effort. Experience shows that those propositions are all relevant to software reuse.

Reuse may generate value from three kinds of improvement: productivity, quality and business performance.

Productivity improvements arise essentially because reuse means less effort through writing less (there are two ways to increase productivity: write faster, or write less, and writing less is easier!). Higher productivity through reduced effort leads in turn to lower development costs and shorter time to market. It can also lead to higher quality (see the next paragraph).

Quality improvements arise in two ways. First, if assets have achieved proven quality in one project, that quality can be carried over to another. Second, if the effort on a project is substantially reduced, there will be fewer defects. Higher quality in turn leads to lower maintenance and service costs, and higher customer satisfaction.

Beware, however! The quality argument has hidden risks. Reusers must not take the original quality of reused assets for granted, nor assume that what constitutes quality in one context necessarily constitutes quality in a different context. Further, substantial reductions in effort may lure reusers

into assuming that the development task is easier than it really is, and may induce carelessness.

Business performance improvements of course include lower costs, shorter time to market, and higher customer satisfaction, which have already been noted under the headings of productivity and quality improvements. They also include improved predictability (smaller amounts of effort are more predictable than larger ones). Such benefits can in turn initiate a virtuous circle of higher profitability, growth, greater competitiveness, increased market share, entry to new markets and so on. These benefits may be direct, if the company's main business is software, or indirect if the software it develops is embedded in its products or supports its business processes.

Let us now look at some examples of estimated benefits that have been reported from systematic software reuse: see *Fig. 1.4*. The examples cover the use of various programming languages, ranging from Ada to Cobol and C++. The examples are presented in alphabetical order of company.

These estimates should be treated with a degree of caution. They all claim improvements in a *key performance indicator*, such as productivity, quality or cycle time; in some cases those improvements are set against a *level of reuse* (which roughly means the proportion of the total work product that was reused). We do not know how those measures were calculated or the reliability of the values obtained. They should therefore be treated as being subject to perhaps substantial margins of error, and as presenting rather rough impressions of achievement. We should not, however, throw out the baby with the bathwater: however error-prone or ill-defined, these impressions are impressive.

The examples compare a situation with reuse against a situation without reuse. The comparison may be done in one of two ways. The first way is to compare the *actual* values of a performance indicator at two points in time, one before the introduction of reuse and one after; note that the length of the time lapse is in no case stated. The second way is to compare the *estimated* value of a performance indicator for a project without reuse against the *actual* value with reuse.

The examples are of claimed results in practice. An alternative way of investigating the potential of reuse is shown in Table 1.1, which sets out a range of *hypothetical* possibilities for the purposes of illustration. It assumes the ability to estimate a *reduction factor* and a *reuse index*. The reduction factor (column 1) is an indicator of what is saved on average (in terms, say, of effort, time or cost) for those work products that exploit reuse. The reuse index (column 2) is an indicator of the proportion of the total work product that exploits reuse: it is closely related to the concept of reuse level as shown in

- DEC
  - *cycle time*: 67%–80% lower (reuse levels 50%–80%)
- First National Bank of Chicago
  - *cycle time*: 67%–80% lower (reuse levels 50%–80%)
- Fujitsu
  - *proportion of projects on schedule*: increased from 20% to 70%
  - *effort to customize package*: reduced from 30 person-months to 4 person-days
- GTE
  - *cost*: $14M lower (reuse level 14%; baseline costs not specified)
- Hewlett-Packard
  - *defects*: 24% and 76% lower (two projects)
  - *productivity*: 40% and 57% higher (same two projects)
  - *time to market*: 42% lower (one of the above two projects)
- NEC (Nippon Electric Company)
  - *productivity*: 6.7 times higher
  - *quality*: 2.8 times better
- Raytheon
  - *productivity*: 50% higher (reuse level 60%)
- Toshiba
  - *defects*: 20%–30% lower (reuse level 60%)
- Sample of 75 projects in 15 companies
  - *quality*: 10 times better (reuse levels 10%–18%)
- Sample of 15 projects in 9 companies (reuse levels up to about 95%)
  - *productivity*: 10 times higher than cross-industry benchmark
  - *time to market*: 70% lower than cross-industry benchmark
  - *cost*: 84% lower than cross-industry benchmark

Fig. 1.4 *Some reported estimates of actual improvements due to reuse.*

Fig 1.4. Multiplying the reduction factor and the reuse index gives an indicator of the broad level of *overall savings* (column 3) that can in principle be achieved, ranging from one-quarter of total development effort, time or cost to nearly three-quarters.

The important thing to realize about this table is that the values shown for both the reduction factors (50 per cent to 80 per cent) and the reuse index (50 per cent to 90 per cent) are known to be achievable in practice. Although it is hypothetical, therefore, it is nevertheless realistic.

It is worth spending some time reflecting carefully on the reported and hypothetical figures presented in Fig. 1.4 and Table 1.1, and asking what the effect

Table 1.1 Illustrations of hypothetical levels of savings.

Reduction factor *(reduction in effort, time or cost achieved for work products that are developed with reuse)* [1]	Reuse index *(proportion of total effort, time or cost attributable to work products that are developed with reuse)* [2]	Overall savings *(saving in total development effort, time or cost achieved as a result of reuse)* [3] = [1] × [2]
50%	50%	25%
50%	70%	35%
50%	90%	45%
60%	50%	30%
60%	70%	42%
60%	90%	54%
70%	50%	35%
70%	70%	49%
70%	90%	63%
80%	50%	40%
80%	70%	56%
80%	90%	72%

would be if you could achieve comparable results in your business. Then you will be ready to ask the next question – whether you are prepared to undertake the substantial, sustained and systematic effort without which such results are not achievable.

1.3 Some Further Introductory Essentials

With the four main ideas about software reuse now in place, we will look in this Section at some other related ideas that are important for completing the outline map of the reuse landscape which Chapter 1 aims to present. They are as follows.

- There are various routes by which assets achieve reuse.
- There is an important relationship between reuse and software process maturity.
- Reuse is an investment, whether or not you call it that.
- Reuse may be pursued within the wider business context of product line practice.

1.3.1 There Are Various Routes by Which Assets Achieve Reuse

This is a surprisingly complicated matter, which we will try to present as simply as possible. The route to reuse is determined by the answers to seven questions, as set out in *Fig. 1.5*.

The answers to those seven questions are in principle independent – meaning there are 288 different routes for an asset to be reused! Some of the combinations, however, are less probable than others. Some comments on the above questions and answers may be helpful.

1 What was the original source of the asset?
- It was developed in-house.
- It was acquired externally.

2 What was the original purpose of the asset (with respect to reuse)?
- It was intended for immediate use on a specific project, without reuse in mind.
- It was intended for immediate use on a specific project, with reuse in mind.
- It was intended not for immediate use on a specific project, but entirely for purposes of reuse.

3 Was the asset reengineered for reuse (prior to its current reuse)?
- It was reengineered for reuse at some earlier time (eg to be put into a reuse repository).
- It has remained unchanged since its original development or acquisition.

4 Was the asset designed for reuse by setting parameters (grey box reuse)?
- It has grey box capability.
- It does not have grey box capability.

5 Was the asset retrieved from a repository for its current reuse?
- It was retrieved from a repository.
- It was in a repository but not retrieved from it.
- It was not in a repository.

6 Was it necessary to 'look inside' the asset in order to assess whether it meets the requirements for reuse (glass box reuse)?
- Glass box reuse was necessary.
- Glass box reuse was not necessary.

7 Was it necessary to reengineer the asset for its current reuse (white box reuse)?
- White box reuse was necessary.
- White box reuse was not necessary.

NOTE: the terms 'glass box', 'grey box' and 'white box' reuse are explained below.

Fig. 1.5 Some of the factors influencing the life history of a reusable asset.

Some people say that buying an asset (question 1) is itself a form of reuse. Their reason is presumably that selling the asset to many customers means it is being reused. That, however, is true only in the trivial sense that selling many cars of the same design represents reuse. It does not, in itself, constitute systematic reuse as defined in this book. Of course, the vendor of the asset may have employed systematic reuse in developing it; and the buyer of the asset may employ it as a reusable asset in a systematic reuse programme. In principle, however, acquisition and reuse are independent concepts, just as we have argued that maintenance and reuse are independent.

A reusable asset may exist in many versions throughout its lifecycle, including being reengineered specifically to make it (more) reusable (questions 3 and 7). This indicates the important need for configuration management in reuse, and for the careful control of maintenance on reusable assets. Reengineering is one of two ways in which an asset may be adapted for reuse; the other is by setting parameters (question 4). Adaptation by setting parameters is only possible if the asset has been designed that way, either originally or by subsequent reengineering. Setting parameters means making selections among predesigned sets of options, so as to determine the exact properties of the asset.

If an asset is reused without the need for any adaptation, that is known as *black box reuse*. If reengineering is necessary, that is to say if it is necessary to change the internal body of an asset in order to obtain the required properties, that is known as *white box reuse*. The intermediate situation, where adaptation is achieved by setting parameters, is known as *grey box reuse*. *Glass box reuse* refers to the situation where it is necessary to 'look inside' an asset, on a 'read-only' basis, in order to discover its properties, in the case where the available description of those properties is inadequate.

A final note concerns the continuing usefulness, or value, of reusable assets. In just the same way that the fitness of software in general is known to decay as a result of increasing age and continuing maintenance, so the fitness of a reusable software asset decays with age and maintenance. The reasons for this are well understood, and derive from the difficulty of maintaining software to keep pace with the rate of change both in user requirements and in technological architectures such as user interfaces and implementation platforms.

1.3.2 There Is an Important Relationship Between Reuse and Software Process Maturity

Software process maturity is a measure of an organisation's capability to produce software to meet goals of quality, cost and schedule. Maturity

increases to the extent that the software process becomes more repeatable, defined, managed, and subject to continual improvement. There are various approaches to measuring software process maturity: the best known is the Capability Maturity Model (CMM) for Software. Initiatives to increase the maturity of an organisation's software process are usually referred to under the umbrella term *software process improvement (SPI)*.

In the software domain, as in other business processes, *process* is conventionally distinguished from *technology*. Process means broadly *what* you do, and technology means broadly the technical practices that determine *how* the process is performed. Thus, the software process refers to activities such as product engineering, project management, quality assurance or configuration management, irrespective of the particular methods and tools (software technology) that may be selected to support those different parts of the process.

A mature process is one that is well managed and continually improving, independently of the technology used to support it. Of course, it is part of a well managed and continually improving process to be sure that appropriate technology is used; but judgments about maturity do not imply judgments about the appropriateness or otherwise of technology.

An important belief of the software process maturity movement is that a technology change on its own does not guarantee improvements in quality, cost, schedule or any other key indicator: the outcome is unpredictable, and change may even make things worse. That belief is clearly justified, for instance, by the relative failure of Computer Aided Software Engineering (CASE) technology. Even so, the more mature your process, the greater are your chances of ensuring that technology changes are assimilated successfully and profitably.

Reuse undoubtedly has a strong technology dimension, in the sense that it normally implies major changes in development and maintenance methods, and will probably need support from specialized tools. Technically oriented staff, such as programmers or software developers/ engineers, may see reuse primarily in such technological terms. But introducing systematic reuse is far more than a change in technology: it must be understood and managed in terms of a major set of changes to the software process. Two Chapters are devoted later in the book to process and management issues; the purpose here is no more than to introduce the essential process-related ideas into this introductory overview of systematic reuse. Those ideas are set out in *Fig. 1.6*.

Systematic reuse is a key practice within the overall software process, and must be treated in the same way as other key practices.

- Goals for reuse must be set.
- Commitment to reuse must be gained.
- Staff must be given the required abilities to perform reuse.
- The detailed activities involved in systematic reuse, and the corresponding work products, must be defined and documented.
- Means of reviewing and measuring the success of reuse must be defined and implemented.

The relationships between systematic reuse and other key practices must be considered, so that reuse becomes an integrated part of the complete process.

- Systematic reuse will impact other key practices, and vice versa. Consider the intimate relationships between reuse and (for instance) requirements management, project planning, configuration management, product engineering, and inspections (or peer reviews).

The relationship between systematic reuse introduction, and any corporate programme of software process improvement (SPI), should be carefully considered and understood.

Reuse does not play a great part in most software process maturity models. That means that, where such a model drives an organisation's SPI programme, there will not be a natural encouragement to include a reuse initiative as part of that programme. Different organisations may handle the relationship in different ways. The following are possible examples, all of which have been observed in practice.

- A SPI programme and a reuse programme may be regarded as distinct major initiatives, each sufficiently large and challenging to have its own goals and support organisation. This case may be described as 'parallel reuse and SPI'. There are two sub-cases, which may be called 'parallel coupled' and 'parallel disjoint', depending on the extent to which they are effectively integrated.
- Reuse may be identified as an improvement priority within an existing SPI programme. This case may be described as 'SPI-driven reuse'.
- Reuse may be identified as the initial driver for major change, and a reuse programme may be launched with little or no allowance made for the principles of successful SPI. If the organisation subsequently discovers and adopts the factors that underlie success in SPI, and ultimately embarks on a wider programme of SPI, this case may be described as 'reuse-driven SPI'. Otherwise it may be described as 'isolated reuse'.

Fig. 1.6 Reuse and the software process.

1.3.3 Reuse Is an Investment, Whether or Not You Call It That

Investment means giving up some benefit in the present, to produce some greater benefit in the future. Undertaking a programme of systematic reuse is undoubtedly an investment, although we may less readily think of it that way than in the case of more conventional investments in land, industrial

plant and tools, buildings, skills, parts inventories, futures, standard manufacturing designs, information and so on. The substantial and sustained management and technical effort involved in introducing reuse over a period of time has a cost, even if it is only the 'opportunity cost' of replacing effort that might otherwise yield more immediate returns; and that cost is incurred in the hope of larger returns in the future. Systematic reuse, as we have seen, involves developing a collection of reusable software assets. The very word *asset* is rich with overtones of investment.

More conventional investments are normally subject to careful and often sophisticated decision making, involving calculated estimates of net present value or payback period, and comparisons with competing investment proposals or with return on investment benchmarks. Actual investment costs are accounted for separately from costs incurred on current account; if benefits and savings can accurately be attributed to a specific investment, then a sound basis exists for computing the actual return on investment after a given period of time.

Software is seldom if ever regarded as an asset that carries value, either by management or in accountancy practice. There is seldom if ever any provision for separating the costs incurred in developing software into investment costs and current costs, or even separating current costs into variable and overhead. Too often all software costs are simply lumped together as overhead. That is one important reason why return on investment figures for improvement initiatives (whether for reuse or for SPI in general) are rarely available and rarely credible.

But let us return to the point that, whether or not we are able to account for it as such, reuse is an investment. Investment unavoidably involves uncertainty, risk and forecasting the unknowable. Even if techniques for forecasting are used, and however smart they are, they cannot remove uncertainty or risk, and successful investment consequently often depends additionally on the subjective element of intuition or flair – characteristics of the entrepreneur. Deciding whether to invest in software in not just an exercise in technical and management analysis: in the end it also needs an element of entrepreneurial instinct.

1.3.4 Reuse May Be Pursued Within the Wider Business Context of Product Line Practice

While relatively new, software product line practice is rapidly attracting the attention of leading-edge software development organisations. According to a frequently used definition, a software product line is *a group of software products sharing a common managed set of features that satisfy specific needs of a selected market or mission.*

Product line practice seeks to leverage the potentially high-payoff technology of reuse by deploying it in the context of a well-defined business strategy and an appropriately modified software process. Application sectors in which PLP applications are to be found include the following:

- TV, VCR, DVD and audio;
- cell phones;
- digital cameras;
- printer peripherals;
- e-commerce systems;
- smart cards;
- geographic information system terminals;
- banking applications;
- telephone switch management and maintenance;
- medical imaging (magnetic resonance imaging, radiography, computer-aided tomography) and medical image archiving and communication;
- car supervision systems, including engine control, parking assistance, pre-crash applications, blind spot detection, autonomous parking, adaptive cruise control and integrated dashboards;
- diesel engine control, for trucks, buses, boats, railroad units, mining and farming equipment, etc.;
- ground vehicle simulators;
- command and control systems and simulators;
- fighter aircraft avionics;
- air traffic control;
- satellite attitude and orbit control;
- elevator control;
- pressure safety release valves and booster pumps.

The above list indicates that product line practice (PLP) is predominantly associated with the development of embedded software (often real-time) for

industrial and defence systems and consumer products. There is a striking absence of representation of 'commercial' software, to support business processes or services, and of 'shrink-wrapped' or 'package' software.

PLP extends the reuse concept, of identifying commonalities and variations in software, upward to the products in which the software is embedded. The opportunities for software reuse influence the specification, design and economics of the products themselves, and vice versa. PLP handles commonalities and variabilities among products explicitly rather than accidentally. It should be requirements-driven, architecture-centric and components-based. A product line architecture is adaptable across the set of products constituting the product line, and should contain change and maximize reuse.

A product line aims to achieve large-grained reuse of assets, and rapid and inexpensive building of high-quality applications within the product line domain using the asset base. It must be able to evolve to incorporate new vendor technology and to provide new functionality required by customers. On their side customers may need to adjust their expectations to fit product line capabilities.

Making decisions about product lines requires understanding their implications on three dimensions: technical (product line architecture, development methods and tools etc.); organisational (team structure, operating models, individual roles and interfaces, communication etc.); business (business goals, divisional charters, market environment etc.).

PLP is driven by the 'produce, consume and customize' principle, which divides software development into two distinct life cycles: domain engineering (which *produces* reusable assets such as requirements, architectures and components, constituting a domain model that captures both the commonalities and the variabilities within the domain), and application engineering (which *consumes* and *customizes* assets to derive individual products within the domain).

Each software development project is important to an organisation, but the project for setting up a product line is far more important. It builds the basis for many specific development projects within the product line. It should be related to overall business and IT strategy, and look several years ahead.

1.4 Systematic Reuse – Crossing Frontiers

The purpose of this Section is to discuss systematic reuse as a critical point of transition in the history of computing, and to discuss why it has taken the software industry so long to achieve what has long been accepted common practice in other industries.

1.4.1 Systematic Reuse as a Critical Transition

We will look at systematic reuse first in the context of the 50-year-long history of software, and then in the context of the 10-year-long history of software process improvement.

Software started in 1948, in the UK. In that year the Manchester 'Baby' was the first machine to demonstrate the execution of stored-program instructions. The following year, in Cambridge, EDSAC was the first stored-program machine to execute a complete program producing 'real-world' results. From the start, the EDSAC group was using subroutines, which can be regarded as an early form of reuse. Reuse was there at the very beginning.

Since then, there has been a continuous stream of innovations that have pushed forward the frontiers of reuse. They include such things as assembly codes, high-level languages, macro-assemblers, code generators, customizable packages, class libraries, test-pack generators and so on. A characteristic of those innovations is that they have been largely on the supply side. They have enabled computer manufacturers and other suppliers to package reusable assets (such as very small chunks of functionality, in the case of high-level languages) in such a way as to raise the plateau on which application developers work, and thus to reduce their workload. Of course, these innovations have been indispensable to the progress of computing, which would have been impossible without them.

On the user side, however, application developers have largely been limited to ad hoc reuse, almost entirely at the code level, using the facilities (subroutines, macros, classes, copy-and-paste, and so on) provided by manufacturers and other vendors. The promise of systematic reuse is that it will enable reuse to 'jump the species barrier' (like mad cow disease!) from the supplier side to the developer side, and bring about the very large improvements in quality, productivity and business benefit that are necessary and possible. That point of crossover, at which we may now be standing, may prove to be a critical frontier in the 50-year history of software, during all of which time reuse has been trying to break through to its full potential.

Let us now turn to the history of software process improvement (SPI), which effectively spans a much shorter period of about 10 years. As we saw earlier in this Chapter, the driving concept in SPI has been process maturity, which simply assesses the degree to which processes are institutionalized, independently of the technical practices which constitute the real fabric of the processes. As we presently understand and use it, SPI offers great benefits in raising us from the swamp of level 1 chaos; but nevertheless it has inherent limitations. Radical progress will come not just from managing the process better, irrespective of its substance, but from improving both substance and management together. Systematic reuse offers arguably the most promising means of improving the substance of the process; pursued in parallel with SPI, it offers exciting prospects of transforming software capability to an extent that SPI could not achieve unaided.

1.4.2 Why So Long?

It may thus be that systematic software reuse by application developers is an idea whose time has finally come. Why, however, has it taken so long? Reuse can be described simply as not reinventing the wheel. The need and the benefits of not reinventing the wheel are understood in all areas of human activity, and much successful ingenuity has been devoted over centuries to achieving that goal. The result has been that as a species we have made progress (according to the saying) by 'standing on our predecessors' shoulders and not their toes'. Except in software. Why?

As a familiar example of reuse, consider automobiles. When developing a new model, a manufacturer may well decide to retain an engine design used in one or more earlier models. The decision whether or not to do so will be influenced by questions such as whether the power production and fuel consumption of the existing engine meet the objectives defined for the new car, and whether its shape fits the new layout.

According to the answers to such questions, it may be possible to use the existing engine design without adaptation. On the other hand, it may turn out that some change is necessary. In that case, further questions will arise about the extent of the adaptations, and how much (for instance) they will incur heavy costs in redesigning the engine production line.

That kind of scenario occurs equally for other parts of a car. It also occurs across other manufactured products, across production plant (such as assembly lines or oil refineries), and across the construction industry. It extends further still, beyond the production of physical artefacts, to commercial services (such as financial products or telecommunications services), internal business processes (such as contracting or double-entry

accounting) and information formats (such as application forms or tabloid newspaper layouts).

It is important to recognize what are being reused in all these cases. They are generalized abstractions – designs, methods, plans, formats – not the specific instances which eventually embody those abstractions. To return to the car engine, it is the design that is reused, by being transferred from one model to another. Reusing an individual physical engine is quite a different thing, and occurs when a second-hand dealer transfers an engine from one car that is to be scrapped to another that is to be sold!

Much is conventionally made of the unique nature of software, and the difference between software and other products. Software, it is said, is abstract, and its production is a design-intensive process. As we have seen, however, reuse normally means precisely the reuse of design-like abstractions. 'Not reinventing the wheel' means not reinventing the concept of the wheel, rather than a specific instance of a wheel. The supposed uniqueness of software does not thus appear to be a terribly good justification for its slow take-up of reuse.

The real difference between software and other products and processes in business is probably a cultural one, which arises from the nature of the product, but which has had very destructive consequences for the management of the process. There are more technical degrees of freedom throughout the development of a software product than exist for other artefacts. Non-software artefacts are constrained by natural laws and the known capacities of the human brain, which reduce the technical options open to designers. It is possible, in contrast, to approach each new software product as a blank sheet, awaiting the full range of the designer's creative inspiration.

A software culture has been allowed to develop in which designers tend to regard their task in just that way, as one of creative inspiration, rather like (for example) artists. Instinctively, they tend to look down on reuse, just as a 'serious' novelist might look down on those authors of cheap romantic fiction who construct storylines out of standard elements. The (not invented here) NIH syndrome is alive and flourishing throughout the software industry, and it is encouraged to thrive by the degrees of freedom that are technically available.

It has also been allowed to thrive by weak management, who generally have had little understanding of the economics of software production, who fail to adapt and apply normal practices of good management, and who are blinded with technicalities by specialists seeking to preserve their vested interests.

Thus reinventing the wheel in software has gone on, and on, and on. We may hope, and indeed with some help from this book, it will not, perhaps, go on much longer.

1.5 A Note on the Experience Base Used in This Book

An important source for the writing of this book has been the real-world experience, elicited through interviews, of a number of organizations that have experimented with introducing reuse into their software development practice. Most of the points made in the book are illustrated by means of *experience notes*, drawn from that repository of practical experience. The repository represents experiences of reuse as seen through the eyes of the people who have really done it. They are the people who know best what they have done, what their successes and problems were, and the business setting into which reuse was introduced. The experience base is an informal summary of these people's stories, of how it seemed to them.

Of the companies that contributed to the experience base, it proved possible to obtain information about four in more depth. These depth studies were used as the basis for comparative case histories, that are presented in parallel in Chapters 8 and 9. These four companies have permitted their identities to be used, and they are referred to by name in both the experience notes and in the case histories: they are Chase Computer Services (UK), ELIOP (Spain), Sodalia (Italy) and Thomson-CSF (France). The other companies remain anonymous.

Reference

1. Reed, D. (1995) Tools for Software Reuse. *Object Magazine* (Feb. 1995).