Flexible Software Development: From Software Architecture to Process

Dharini Balasubramaniam, Ron Morrison
School of Computer Science
University of St Andrews, UK
{dharini, ron}@dcs.st-and.ac.uk

R Mark Greenwood, Brian Warboys
School of Computer Science
University of Manchester, UK
{markg, brian}@cs.man.ac.uk

Abstract

Modern software development practices show that there is significant diversity in the product, process and geographical location of software development due to economical, technical and logistical constraints. Such diverse development demands flexibility in the software engineering methodology. In this paper, we propose the Cellular Development Methodology (CDM), an architecture-driven approach to flexible software development. CDM derives the development process and its architecture from the software architecture of the product. The software development process is structured and customised as a network of cooperating cells. Our contribution is a cellular approach to software development based on the software architecture of the end product whereby a network of configurable cells yield customisable, potentially distributed software development processes which in turn produce software that is highly tailored to user requirements.

1. Introduction

Modern software development practices [10] indicate that software is developed in terms of components, either COTS or custom-produced, which are composed to create the final product. Furthermore, the software development process and its products consist of units of different granularity, possibly distributed across the globe. Different phases of the development process utilise different tools on different platforms and may be carried out by different teams in different locations. Such diversity in product, process and location may be due to economical, technical or logistical constraints. These diverse development environments and release platforms demand flexibility in the software engineering methodology.

Flexible software development methodologies allow the development process to be configured to the characteristics of the specific system and the skills of the developers involved. We take the approach that, irrespective of their cause, variable granularity and geographical distribution can be used as an opportunity for flexible development. Given well-defined abstractions and interactions across boundaries of geography, technology and expertise, a flexible methodology can allow processes to be personalised to development teams and product components.

However, flexible development requires mechanisms to ensure that coherent final systems are produced despite potential differences in the processes and teams that create their components. Software architectures [22, 25] can play an important role in maintaining such consistency across processes and products since they specify constraints over the structure and behaviour of systems. The software architecture provides a basis for creating a flexible development process though the architecture itself may be one of the products of the development process. Here we propose a software methodology, Cellular Development Methodology (CDM), which provides the flexibility to structure and customise software development by deriving the architecture of the process and the process itself from the software architecture of the product.

The software architecture of the product as well as the design decisions taken during product development are reflected in the architecture of the development process. In CDM, the software architecture of the product is expressed in terms of components and their interactions and the process architecture is expressed in terms of cells and their operations. Just as components are composed to form software systems, individual processes for developing components may be composed to form system development processes. In CDM, each component has a process which drives its development. The flexibility of the approach stems from the facts that the development process for each component is captured in a cell with well-defined semantics for operations permitted on the cell and that each cell provides the infrastructure for customising these operations. The architecture of the overall
process is initialised as a single cell representing the development of the whole system and is refined into a network of cells by cell operations guided by the software architecture of the product. The network of customised processes in the cells, composed by the semantics of the corresponding cell operations, mirrors the software architecture and forms the development process for the product. Thus each component is developed in a flexible manner while still existing in the context of other components as specified by the software architecture of the overall system.

Our contribution is a cellular approach to software development based on the software architecture of the end product whereby a network of configurable cells yield customisable, potentially distributed software development processes which in turn produce software highly tailored to user requirements.

### 2. Flexible development and architectures

The software engineering community has proposed a number of development methodologies to provide flexibility where requirements and available software components can change rapidly.

The Rational Unified Process (RUP) is an instance of the more generic Unified Software Development Process based on UML notations [13]. The RUP can be customised to a specific project or organisation through best practices, a process library and a configurable architecture.

The advocates of agile development propose more radical changes to traditional development processes. Extreme Programming [1] and Scrum [2] are two of the best-known agile development processes. They are lightweight, iterative and incremental processes which emphasise the need for communication and collaboration between software developers and frequent customer interaction to build confidence in the value of the software. Extreme Programming (XP) recommends short development/release cycles, incremental planning, evolutionary design, automated testing and flexible scheduling to provide adaptability. Scrum is designed to reduce development time so that the resulting product is still relevant to customer requirements. It takes an all-at-once view to software development where many of the traditional phases of the software lifecycle are carried out at the same time by a small, dedicated development team.

The Twin Peaks development model [16] gives equal status to requirements and architectures and is driven by three concerns common to flexible development approaches, namely IKIWISI (I’ll Know It When I See It), COTS (Commercial off-the-shelf) software, and rapid change [3]. IKIWISI captures the fact that customers’ requirements emerge only after significant analysis and feedback. The importance of COTS comes from the fact that identifying, selecting and adopting existing software packages, or components, is an increasing part of software development.

Component-Oriented Rapid Development (CORD) [23] is an approach targeted mainly at small systems for highly specific applications. It is similar to XP in basic concepts but uses the component structure of an application for parallel, distributed and interactive development.

Agile processes are tailored to maximise sensitivity to change. While the RUP is described as architecture-centric, its customisation is based on the project and the organisation rather than the architecture of the product. Although CORD uses the structure of applications to distribute development, the units of distribution are not flexible. Extreme Programming and Scrum concentrate on human, organisational and time factors. While they contribute to agility in general, they do not explicitly utilise the software architecture of the product.

CDM can be considered as complimentary to those above. It aims to provide a holistic approach to flexible software development through potentially distributed, multi-process structuring. It emphasises the structure and distribution of both products and processes. Existing methodologies and techniques can be integrated within the CDM framework to deal with specific aspects of software development. Mechanisms for addressing specific concerns in development are orthogonal to the overall philosophy of CDM. For example, all the methods described earlier in this section can be used as cell customisations in CDM, as will be illustrated later.

CDM is concerned with architecting the development process and not with the execution semantics of the product. While developers may choose to make the architecture of the development process match the execution architecture of the product, CDM does not impose that association. For example, the fact that the development process for a system is distributed across a number of sites does not imply that the execution of the resulting product is distributed.

### 3. Structuring software development

Central to CDM is the notion of a process architecture consisting of a network of cooperating software development cells derived from the software architecture of the product. The development process is
composed of process fragments from the network of

3.1 Products, processes and architectures

In a simple architecture-based approach to software
development, a process first produces the software
architecture of the product, and consequently develops
the product guided by this architecture. The execution
or testing of the product provides feedback to the
process, which evolves the architecture and the product
as required. Figure 1 shows the more intricate
relationships between product, process and their
architectures in CDM.

Figure 1: Product, process and architectures

CDM makes use of a process architecture as shown
in Figure 1. Here Process-1 produces the software
architecture of the product (Product Architecture),
which is then used by Process-2 to create an
architecture for the process (Process Architecture).
Process-3 uses the Process Architecture to define a
customised Process-4 which creates the Product
in compliance with the Product Architecture. Process-4 is
shown as both a product and a process to denote that it
is created as the result of another process. Any
feedback provided by the product to Process-1 will
result in another iteration of the cycle. Although, for
clarity, the processes involved are shown as separate
entities, in practice they may be sub-processes of a
complex process or a single, adaptable process.

Figure 1 shows the relationships amongst the
aggregate process(es), product and architectures
associated with software development using CDM.
These are divided among the cells. The process
architecture is the network of cells. The product and
the process are the composition of the products
produced by each cell and the process fragments within
each cell, respectively.

CDM does not impose the use of a specific
architecture description language or framework.
However it assumes that an architecture specification
of a system consists of at least the following
information:

- Structure of the system in terms of collaborating
  components
- Behaviour of the system including interactions
  between components
- Constraints over the structure and the behaviour
  of the system
- Design decisions about the product
development

3.2 Cell infrastructure

Each cell in the process architecture maintains a
reference to the features (such as component,
interactions and constraints) associated with the cell in
the software architecture of the product. A cell has a
product part (what is produced) and a process part
(how it is produced). It produces and evolves a
software artefact guided by its local development
process in accordance with the associated software
architecture specification. Within a software
development environment (SDE) that supports CDM,
each cell acts as a customisable context for a well-
defined part of the overall product.

In keeping with lightweight development trends, we
adopt a just-enough approach to the provision of cell
infrastructure. Cells are initialised with minimal
content but cell parameters allow their processes to be
customised to the product and its context. This practice
also allows development decisions regarding a cell to
be postponed until all required knowledge has been
gathered without necessarily delaying progress on
other cells.

The process in this context is inclusive of the people
and tools involved. Cell parameters may describe the
types of development tools or services and the roles of
different people involved in creating the cell’s product.
Specific individuals and tools may not be known on
cell creation and can be dynamically bound to the
process when required.

A cell has significant autonomous ability to
reconfigure its process, and thus continue to specialise
itself as more is known about the development context
and product. The network of cells gives a
compositional software development approach that is
flexible both in the creation of cells and in the process
within each cell. Their high cohesion and low coupling
make cells an effective unit of distribution.

3.3 Cell operations and network

The process part of a cell is based on a set of
customisable operations. These can be classified into
internal operations that alter the content of a cell and external operations that affect the cell network. From software methods developed over the last 20 years and our own experience [28, 15], we have identified the following primitive operations:

- **Produce** is the only internal operation and creates the cell product.
- **Decompose** divides a cell into a set of separate sub-cells yielding a hierarchy for understanding and managing the complexity of a large system [26]. The decomposition is performed according to the constituent structure of the corresponding architectural component.
- **Refine** creates a more concrete version of a cell. It can be used where there is an established sequence of artefacts between an initial design and a usable product component. The refine operation is also used where a required component can be obtained through the reuse of an existing component, which may be bought from a third party, chosen from a library of components or generated. The refinement is carried out according to the behaviour and the design decisions specified for the corresponding architectural component.
- **Verify** checks that the cell conforms to a set of constraints. This covers a spectrum of methods including formal verification techniques such as model checking and theorem proving and other checks such as code inspection, unit testing and customer feedback. This operation results in a new cell annotated with the verification status of the original cell. The verification is performed to satisfy the constraints specified for the corresponding architectural component.

New cells are created through external operations within the process of an existing cell. The parent and child cells are related by the appropriate decompose/refine/verify links. The functionality of a leaf cell is captured in its entirety by its product. A cell with one or more child cells utilises their products in accordance with the semantics of the external operations which created the child cells, which in turn are guided by the software architecture specification associated with the parent cell.

Cells at the higher levels of the decomposition hierarchy in a network are typically abstract; their products express how the products of their sub-processes are composed to satisfy the required architectural behaviours and constraints. When a cell is created, its product is initialised to be the software architecture specification of its product and consequently refined to incorporate further computation.

Figure 2 below shows an example of a cell network modelling a process architecture.

In Figure 2, cell S is the starting point of development. It is decomposed into two cells, S.1 and S.2, by its process. Cell S.2 is refined into cell S.2-1 by adding more detail. Cell S.1 is further decomposed into cells S.1.2 and S.1.2. Cell S.1.2 is verified by some mechanism to create cell S.1.2'. Feedback from the verification is used to refine this cell into cell S.1.2'-1.

### 3.4 Feedback and adaptation of the cell network

An existing cell in the network may have to be removed during a flexible development process. Cells can provide an explicit **delete** operation for this purpose. However the **refine** operation can also achieve the same result by refining the unwanted cell into a **null** cell. This has the advantage of preserving the cell information for future needs without it having to be part of the current system. Software and process reuse is catered for by the **refine** operation linking to an existing cell which can provide the required functionality.

There are two kinds of feedback available on the development process. The **verify** operation provides feedback at the cell level, as a result of carrying out automated testing, model checking or product demonstration to customers. In addition, the cell network itself forms feedback at the system level since it reflects the up-to-date structure of the product and its development.

The result is a development approach that starts with a single cell representing the development of the whole system and grows as development progresses. The network continually evolves as more decisions are made.
made, more detail is added and more feedback is received. CDM does not impose a specific granularity for cells or a required level of detail for operations.

The cellular nature of CDM enables a variety of information about the development to be extracted using different views over the cell network. The *decompose* links provide a tree representation of the development based on the software architecture of the product. This information may be used for consistency checks between product and process structures. The *refine* links represent the evolutionary development of components. As the *refine* links can introduce sharing through reuse, the overall network is a graph rather than a tree. The *verify* links provide additional information about the status of specific cells. For example, they can determine whether a cell conforms to the constraints specified in the associated software architecture of the product. Together, these links provide a history of the development process, and consequently some of the key decisions made during development.

CDM is not dependent on specific technologies but provides an architecture-driven model of the cell network to support the development processes. Augmenting this model with support for the operations and parameterisation gives a configurable software development environment that maintains links between the software artefacts, their architectures and the processes that produce them.

### 3.5 Access to product architecture

The software architecture of the system, which determines the architecture of the development process, may be known in advance, particularly where a family of products is created with shared characteristics. In this case the product of the first cell is initialised to be the known software architecture of the system. This information is used to create further cells thorough decomposition and the relevant part of the architecture is passed on to the newly created cells. If the product does not belong to an existing architecture family, then its software architecture may be defined as part of the development process. We envisage two possible scenarios for architecture definition:

- **Eager**: An iteration of the development process outlined above with a software architecture specification as the product prior to the development of the target system itself.
- **Lazy**: Concurrent development of the software architecture and the system whereby the process in each cell consists of sub-processes to produce the architecture and then the product. In this scenario a cell only has to decide what its sub-components will be, and how they will interact. The responsibility for deciding the internal architecture for each sub-component can be devolved to its cell.

In the former scenario, an architecture specification for the system is created by a network of cells using a meta-architecture to structure the process. The meta-architecture contains information on what constitutes an architecture specification and thus may be fixed for a specific ADL or architecture domain. The decomposition hierarchy of the process models the component structure while the refinement and verification links lead to more complete and correct specifications of components. The resulting architecture is then used to drive the development of the system through another iteration of the methodology.

The latter case is of more interest where agile development techniques are employed. The initial cell takes the requirements of the product and creates the most abstract versions of the software architecture of the product, the architecture of the process and the process itself. Cell operations then recursively build up a network of cells, producing the corresponding fragments of product architecture, process architecture and process with every application.

### 4. Example

To illustrate CDM, we consider a process for the development of a new electronic point of sale (EPOS) system for a chain of stores. It is developed in a software development environment using CDM. We assume the existence of a software architecture for the EPOS system, as shown in diagrammatic form in Figure 3.

![Figure 3: Architecture of the EPOS system](image)
each back office component provides a set of services and all of these can be accessed by any front-office component. In addition, there may also be constraints, for example that front-office components have access to both a primary and a backup version of all the services provided by back-office components.

This architecture is used to derive the process for developing the EPOS system. The starting point in this example is a cell representing the development process for the EPOS system. The EPOS cell is initially created with default operations, customisation facilities and a link to the EPOS software architecture which becomes the first version of the EPOS product. The process in the EPOS cell customises the EPOS cell operations according to the software architecture. For example, the decompose operation will now create two new cells corresponding to the development of front and back office components. The process in the EPOS cell then applies the decompose operation. Thus the EPOS cell is decomposed into front and back office cells. By recursive applications of the same methodology, the front office cell is further decomposed into cashier authentication, cash management and till transactions cells and the back office into price lookup, credit/debit card operations and stock control cells. This yields an initial network structure of nine cells in a three-level decomposition hierarchy.

As cells are supported by processes, we can revisit a cell and continue to add additional cells or refine existing ones. The cell structure reflects key decisions about the EPOS system architecture and its development. The resulting cell network is shown in Figure 4 below.

For the till transactions cell, an agile XP approach, with Java as the implementation language, is chosen. The refine operation of this cell is therefore customised to create a new cell which uses a process based on XP and applied. In contrast, for the price lookup cell a more structured development process is selected, and for efficiency reasons, C++ is the recommended language. This cell will be the origin of a set of cells linked by refine, where each cell contains the process for developing an artefact described in the structured development. The first choice for the card operations cell is to reuse a component from an existing system, and the cell is parameterised with a process based on selecting and testing a third-party component.

The composition of the processes from the cell network according to the semantics of cell creation results in a flexible development process for the EPOS system that matches the selected architecture.

5. Distribution of the software process

The cellular approach of CDM captures the logical separation of the development processes of components through the decomposition hierarchy of cells. The logical distribution of components is captured implicitly in configuration information of cells, which expresses how a cell creates its product using the products of its children. Software architectures can be used to capture high-level decisions about the physical distribution of system components. The cellular approach of CDM can aid the physical distribution of development amongst a number of sites, and where appropriate, this process distribution will correspond to the physical distribution decisions in the software architecture of the product. Distributing the cell network into a number of sub-networks in distinct SDEs, at separate sites, works best when there is loose coupling between components in different sub-networks.

While distribution contributes to the autonomy and the flexible development of cells, it also raises a number of issues. A system generally begins as a single cell at a single site. As the development process progresses, new cells will be created and may need to migrate to other sites to reflect architectural decisions and/or exploit development expertise. This movement has implications for reuse and sharing. If a cell moves from its original site, the links to and from this cell can be severed and rebound at the new site to any depth of the transitive closure. Alternatively the cell can maintain its links to the original site by replacing local references with absolute addresses. Developers may choose to employ a combination of these strategies for the links of a single cell depending on application requirements. They must also decide whether the links are updated in an eager (as soon as the cell moves) or lazy (only when the link is referenced) fashion.

In the EPOS example, the price lookup cell is moved to a new site with its own SDE. However it must be reachable from the back-office cell at the...
original site in order to maintain the decomposition hierarchy. Processes are generally rebound to equivalent ones within the new site’s SDE. While most processes at distributed sites will be distinct, reflecting the local autonomy and particular skills of the sites, particular process fragments can be adopted throughout a system and handled as shared resources. In the example, there may be a single process fragment for validating the user interface style of all custom-built components.

Figure 5 illustrates the distribution of the EPOS development described earlier. For clarity, only those cells corresponding to the back-office subsystem are shown.

![Figure 5: Distributed EPOS development](image)

The development of the price lookup cell has been moved from site 1 to site 2. The back-office cell now has two local and one remote decompose links. At site 2, cell Price is refined to Price – RUP. Both these cells make use of one of the processes from the local process library. In addition, the refined cell has a remote link to a process from the library at Site 1, which is shared by the Cards cell at the original site.

The process for each cell is developed independently from others. If processes from different cells need to interact, they exploit synchronisation mechanisms provided by the SDE to coordinate their collaboration.

**6. Related and future work**

The ArchWare Tower [30], developed during the ArchWare project [17, 31], implements a specialisation of CDM in the form of a cellular SDE. The network of cells constituting the development environment is derived from the software architecture of the target system. The tower has been built to support the architecture-driven development of (dynamically) evolvable software. Thus the cellular structure of the process is closely tied to the execution architecture of the product. The SDE is released as part of the product to support evolution after deployment. During the ArchWare project, the generic tower was customised to create different specialised instances of the SDE for use by the two industrial partners in the project. Each of these companies further customised their towers as required to cater for individual customers. This experience with the ArchWare Tower shows that the cellular structure of development along with customisable operations of cells provides a basis for flexible, incremental development and the independent dynamic evolution of components.

The ArchWare Tower itself builds upon previous work on the development of large long-lived software systems. The CADES development environment [21] for ICL’s VME operating system [29] was based on state-of-the-art software engineering in terms of module decomposition [19, 20, 26] but rather inflexible in terms of accommodating new development techniques. This was a major influence in the design of the process-centred software engineering environment IPSE2.5 [28], and research combining formal specification and an evolutionary approach to software design [9]. The problems of designing large process models led to the use of software architecture [17] and exploiting persistent links between software components [15]. In the ArchWare Tower, the module decomposition document of Parnas [20] has been replaced with the software architecture model expressed as a network of cells; rather than reading an overview document developers browse the development environment to learn how different parts of a system fit together.

The CDM framework presented in this paper concentrates on the development of a system. This framework can be extended to link executing systems with their development environments through a programmable interface, enabling dynamic evolution of software, process, software architecture and environment in step with one another. The Meta-programming Framework [14] has been developed to provide such an interface for the ArchWare system.

While the ArchWare Tower is a very specific implementation, CDM recognises the general benefits of using architectures to help developers manage the complexity of software systems and integrates this with the use of hierarchy to manage large systems [9, 28]. A common thread in these research areas is the exploitation of software architecture to provide developers with a good overview of the context of their work while also preserving their flexibility to choose the most appropriate development processes for their components.

One of the features of CDM is using software architecture to guide the loose coupling of development processes, allowing different cells to specialise their development processes and develop at different rates. Each cell has access to a library of
process fragments that it can combine, specialise and use. This facility can be extended to give each cell access to a library of reusable architecture resources, for example architectural styles and patterns, and related process fragments for the resources. While work on the ArchWare project has shown that it is possible to express process and product resources in the same ADL, the strengths and weaknesses of other ADLs in this context requires further research.

Recently, there has been a recognition that ADLs have concentrated on structure and behaviour of systems at the expense of design rationale [4, 7]. CDM provides the opportunity to record rationale not just in the software architecture of the product, but also by keeping historical information about the cell development process.

The IEEE 1471-2000 provides a definition of architecture that explicitly refers to evolution:

“The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution” [11]

There has been significant research in the use of software architecture of products for both design-time and run-time evolution [5, 8, 18, 24], and managing the evolution of product architectures [27]. One difference between these and CDM is that a global architecture model is used to guide evolution, and the emphasis is on the automatic application of pre-determined evolutions. However, the separation between architectural evolution management (maintaining the consistency between the architectural model and the implementation) and architectural adaptation management (deciding and enacting architectural changes) [8, 18] is a feature that can be incorporated into CDM to specifically target self-adaptive systems.

Autonomics is an area that has adopted a cellular approach to address the complexity of systems management [12]. In autonomics systems the autonomic manager and the managed component correspond to the process and product in CDM. While autonomics systems aim for self-management, CDM aims to provide an effective context for developers. The incorporation of autonomic features into CDM offers the potential of a spectrum between automatic and developer-provided changes. One area of particular interest is the automatic checking of architectural constraints and notification of consistency problems.

7. Conclusions

There is considerable demand for flexible software development processes which allow for diverse business constraints to be met within the same product. An appropriate structuring mechanism for both process and product enables the distribution of the development process, creating more opportunities for flexibility at local and global levels.

We have presented Cellular Development Methodology (CDM), a flexible, architecture-driven approach to software development. It models the architecture of the development process as a network of cells, based on the software architecture of the product. The cells link software artefacts to the processes that develop them. Parameterisation of cells allows processes to be customised to product and context. The cellular structure also facilitates distribution leading to better customisation, greater parallelism and hence more agile development.

A software development environment supporting CDM provides developers with the appropriate context when working on a specific cell and also an overall view of the development, libraries of artefacts and processes and mechanisms for distribution. The cellular approach can also enable the extraction of useful views from the network of cells, corresponding to different styles of development. CDM does not prescribe any particular architecture description language, nor does it impose a specific development model. It concentrates on architecting flexible and distributed processes through establishing a network of cells that can be customised. Thus development processes are made compliant to the needs of their products and context. The coherency of the cell network is maintained by cell operations changing the network so that it conforms to the constraints specified in the software architecture of the product.

There is an interesting tension between CDM and Conway’s Law [6], which states that “organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations”. By contrast, CDM concentrates on the software architecture to drive the product development. However, the cost of development may be crucial to product delivery and economical, technical and logistical constraints may influence the choice of cell structuring. How much of such cell structuring is due to the accident of the organisational structure will vary on a cost basis from product to product.

8. Acknowledgements

This work was partly supported by the EC Framework V project ArchWare (IST-2001-32360). It builds on earlier work in compliant systems architectures funded by the UK Engineering and
Physical Sciences Research Council (EPSRC) under grants GR/M88938 and GR/M88945.

We also acknowledge the conference reviewers whose comments provided valuable assistance in improving the paper.

9. References


