ApleT is a framework for performing agile product line engineering, particularly with .NET web applications and a behaviour-driven development approach in mind. ApleT follows the feature-oriented software development (FOSD) methodology.

ApleT is currently in very early stages of development. See django-productline for a much more mature FOSD framework for Django and Python.

Software product line engineering (SPLE) provides an efficient means to produce software that is delivered to multiple customers, in bespoke form, derived from a composable, reusable set of assets. Effective use of SPLE can reduce defects, costs, and time to market by a factor of 10 or more. SPLE is desirable in small- to medium-sized enterprises, where a common occurrence is the introduction of a one-off product that proves to be popular, and over time requires multiple customer-specific versions. However, in its more traditional form (the proactive approach, with a focus on the long-term up-front domain engineering phase), SPLE is not always viable for SMEs. Agile product line engineering (APLE), the combination of SPLE and agile software development, is a reactive approach to facilitating SPLE in SMEs that can cut down the long-term investment of the domain engineering phase.

Behaviour-driven development (BDD) is an agile software development methodology focused on specifying and testing software through business-focused examples. Through incorporating the concept of variability management, a core tenet of SPLE, into behaviour-driven development, ApleT provides an agile approach to product lines that provides automated, specification-based regression testing across the entire product line. This approach enables the safe evolution of product lines both in terms of requirements and in terms of architecture that avoids big design up front.

FOSD is a particular approach to SPLE that focuses strongly on the idea of features and their representation throughout the entire software lifecycle.
Why do we need product line engineering for web applications?

**THE FOLLOWING STORY** has happened thousands of times. A company puts a product on the market, and the product proves to be very successful. Customers use the product and it’s almost perfect, but it needs some changes to make it really fit the context in which it’s used. The company considers this and provides a customer-specific version. At the same time, on the basis of customer feedback, the company realizes that several customer segments would be better served with a product focused for each segment. So, the company ends up with a multitude of significantly similar product versions.

At this point, the company realizes that many of the required changes must be implemented for most or even all product versions and that implementing the same change multiple times is really inefficient, time-consuming, and error prone. This often results in the creation of a platform from which the different products and customer-specific versions can be derived. This significantly improves development efficiency. However, a new challenge enters the arena: managing the points in the platform where the product versions’ functionalities differ — that is, variability management.

—*Bosch2015*

Why do we need to incorporate behaviour-driven development?

...much of the tremendous power of features is yet be unlocked by making features explicit throughout the entire systems and software lifecycle

—Krzysztof Czarnecki, foreword to *Feature-Oriented Software Product Lines* [Apel2013]

The explicit treatment of features in requirements, architecture, implementation, and verification and validation can greatly improve the management of software.

—Krzysztof Czarnecki, foreword to *Feature-Oriented Software Product Lines* [Apel2013]

We wish to combine the benefits of BDD, SPLE and FOSD.

“the idea of variability-aware analysis is not to invent new kinds of analysis techniques, but to lift existing analysis techniques developed for individual programs to entire product lines (that is, to domain artifacts).”

—Feature-Oriented Software Product Line, p.261 [Apel2013]
Software product line engineering

Software product line engineering (SPLE) provides an efficient means to produce software that is delivered to multiple customers, in bespoke form, derived from a composable, reusable set of assets. Effective use of SPLE can reduce defects, costs, and time to market by a factor of 10 or more. SPLE is desirable in small- to medium-sized enterprises, where a common occurrence is the introduction of a one-off product that proves to be popular, and over time requires multiple customer-specific versions. However, in its more traditional form (the proactive approach, with a focus on the long-term up-front domain engineering phase), SPLE is not always viable for SMEs. Agile product line engineering (APLE), the combination of SPLE and agile software development, is a reactive approach to facilitating SPLE in SMEs that can cut down the long-term investment of the domain engineering phase.

2.1 Overview

Software product line engineering (SPLE) is a systematic approach to achieving software reuse. The main goal of SPLE is to enable software vendors to efficiently tailor software products to the requirements of individual customers. It aims to minimise the overheads incurred when building a family of software products that have significant shared features, yet various differences. SPLE and its reuse-oriented methodology has many advantages, both financial and technical, and a number of industrial case studies document the successes it can bring.

2.1.1 Hand-crafting vs mass customization

The analogy from which SPLE takes its name is that of product lines in manufacturing, such as those found in the automobile industry. Before the advent of mass production in traditional manufacturing, goods were hand-crafted – each item was produced individually, in a time-consuming and labour intensive manner, but nonetheless one that allowed for individual tailoring. The onset of industrialisation brought with it mass production – whereby, through standardization of parts and processes, the time-to-market of an individual product was greatly reduced. This came with a loss of customisation, however. To overcome this, the idea of mass customization was introduced. Through the intelligent use of reusable core components and points where variations can be introduced, a wider portfolio of products can be produced while still reaping the benefits of mass production. In this context, a product line is a family of related products that can be built from reusable parts, and which have many commonalities with some variabilities between them.

A similar history of early hand-crafting, followed by a move to mass production, followed by the need for mass customization, can be seen in software development. Software was originally hand-crafted to companies and to hardware. As the demand for computers grew, standard software was produced that could run on multiple architectures, which as with manufacturing reduced the temporal and financial overheads of bespoke tailoring. The parallel of the need for mass customization arises, however, with customers either unable or undesiring to run exactly the same software.
Efficiently produced, but individually customised, software was required. SPLE is the field of research and practice that has grown to meet that requirement.

2.1.2 SPLE history

The idea of software product families was first documented by Parnas [Parnas1976]. SPLE has been researched in increasing depth over the past two decades. Its use in industry has also increased, beginning with larger businesses such as Boeing, Hewlett-Packard, and LG, and latterly making inroads into smaller enterprises [Pohl2005]. It is by no means only for commercial usage – the open-source Linux kernel, for example, which must run on a vast array of hardwares and architectures, is one of the largest software product lines in existence.

2.1.3 Benefits

The key benefits of SPLE are the reduced cost of new products, the reduced time to market of new products, the improved quality of products, and the facilitation of bespoke software. There are overheads associated with SPLE, however, which include the cost of the initial planning of the product line, and the extra complexity required in managing variability. These overheads, in combination with its relative newness, mean SPLE is not yet in widespread use in small- to medium-sized enterprises (SMEs) [Laguna2009]. Efforts exist to bring the benefits of SPLE to companies less able to accomodate its large up-front analysis times [Krueger2002a] [Ghanam2009] [McGregor2008].

2.2 Variability management

2.2.1 Variability model

[Todo]

variability model

2.3 Binding times

The binding time is at which point the mechanism for distinguishing between variants of a feature come into play. The binding time might be compile-time, load-time or run-time.

An example of compile-time binding would be preprocessor directives, that conditionally include or exclude code at the point of compiling the application.

An example of a load-time binding implementation mechanism would be conditional execution of code via parameterisation, where the selected value of the parameter is loading from a configuration file.

Run-time binding can take into account real-time changes in the context of the application.

2.4 Artifact dependencies

[Pohl2005] (p.83)

[Todo]
2.5 SPLE Process

SPLE is commonly split into domain engineering and application engineering. A traditional process model for SPLE [Apel2013] breaks the work into a two-dimensional space, with problem space and solution space making up one axis, and domain engineering and application engineering making up the other, and with four main clusters of tasks in traditional SPLE are:

- Domain analysis - this is requirements engineering for an entire product line. The results of domain analysis are often documented in a feature model `<feature-model>`.
- Requirements analysis - the needs of a specific customer as part of application engineering
- Domain implementation - developing reusable artifacts that correspond to the features identified in domain analysis
- Product derivation - the production step of application engineering, where reusable artifacts are combined according to the results of requirement

2.5.1 Types of process

There are three broad ways in which an organisation can undertake product line engineering: these are the proactive, extractive, and reactive approaches [Krueger2002a].

- Proactive: The proactive approach is the more traditional approach to SPLE, where complete domain analysis and variability management is performed in advance of any application engineering. In many small- to medium-sized enterprises the large upfront costs of this approach mean it is not viable [Buhrdorf2004].
- Reactive: In contrast, the reactive approach involves creating and updating the product line as and when new members of the family appear. Extensive domain analysis is avoided, and time can be saved.
- Extractive: The extractive approach involves bootstrapping a product line by taking existing products as the base for the core assets.

The extractive and reactive approaches can be used together.
Behaviour-driven development (BDD) is an agile software development methodology focused on specifying and testing software through business-focused examples.

### 3.1 Gherkin

Gherkin is a business-readable DSL for specifying system behaviour.

An example of feature specifications in a Gherkin feature file looks as follows (excellent ninja example from http://pythonhosted.org/behave/tutorial.html#feature-files):

```
Feature: Fight or flight

In order to increase the ninja survival rate,
As a ninja commander
I want my ninjas to decide whether to take on an opponent based on their skill levels

Scenario: Weaker opponent
  Given the ninja has a third level black-belt
  When attacked by a samurai
  Then the ninja should engage the opponent

Scenario: Stronger opponent
  Given the ninja has a third level black-belt
  When attacked by Chuck Norris
  Then the ninja should run for his life
```

Gherkin files describe a feature with a name and description. These features are broken down into scenarios, which are further broken down into Given, When and Then steps.

There are many more components to a Gherkin feature file. For more information please see the Gherkin reference.

### 3.2 Step definition code

Step definition code is a layer of code that is automatically executed when the specification tests are run.

Each individual step maps to a method within the step definition code. When a scenario within a feature that contains the corresponding step for the step definition is run, this will execute the method within the step definition code.
3.3 Progress reports

When specification tests are run, progress reports are produced. These reports show the status of the implementation of the features within the product.

Todo
image of progress report

Todo
pickles
Feature-oriented software development

FOSD is a particular approach to SPLA that focuses strongly on the idea of features and their representation throughout the entire software lifecycle.

The book Feature-Oriented Software Product Lines: Concepts and Implementation is the best introduction to the topic. There is a paper available online from the Journal of Object Technology that gives an overview of feature-oriented software development [Apel2009].
Ghanam [Ghanam2010a] outlines benefits of bringing the use of executable acceptance tests into the product line paradigm.

- Consistency between the feature model and the code artifacts
- Continuous two-way feedback
- Exploiting hidden variability concerns

References
Linking feature model features to BDD features

We need to discuss the different ways in which we might map between features on a variability model, and features as we have them in BDD.

First of all, we need to define what we mean by both.

The word feature is a nebulous term that has been given many definitions. Here we use the definition from [Apel2013]: a feature is a characteristic or end-user-visible behavior of a software system. This definition fits well with the notion of behaviour-driven development. However, we must define a key distinction between the features of a variability model (we’ll call these VM features) and the features within BDD (BDD features).

In ApleT, BDD features are artifact dependencies of VM features. The role of the variability model is to define variability, and the role of the BDD features is to define expected behaviour. As we would not expect the VM to contain features for every code artifact (e.g. classes and methods), we should not expect the VM to contain a feature for every possible test artifact. That is to say, each VM feature can map to one or more BDD artifacts. We must define at what granularity this mapping occurs. Intuitively, based on nomenclature alone, we may feel that there is a one-to-one mapping between VM and BDD features. However, we do not wish to have a full BDD feature file for each VM feature. In the simplest case we could do this if required - but this may mean duplication of BDD test artifacts. (For example, for two variants of control flow, we would need a complete BDD feature file.) The less variability-awareness we have in our test artifacts, the higher the level of granularity we must fix our artifact dependencies.

There could, for example, be a 1-to-1 correspondence between a FM feature and a BDD feature. This, however, would lose the fidelity of variability within a BDD feature. Selecting variants of FM features would select different BDD features. Our test artifacts would not be variability-aware by control flow or by data. To vary these aspects, we would need to clone-and-own scenarios and steps. To enable variability by control flow without test artifact duplication, we must have VM features map to scenario steps selections. To enable variability by data without test artifact duplication, we must have variants map to BDD data selections - if it is discrete parameterisation then through alternative features, it parameterisation of continuous values then through some other mechanism such as feature attributes.

“We accomplish this by relating use cases, use case scenarios and use case scenario steps to features of appropriate types in a feature model” [Eriksson2006]

The PLUSS approach provides advice on mapping use cases to features [Eriksson2005]. One or more use cases can map to a feature. One or more scenarios can map to a feature. One or more can relate to any feature on the feature model. Cross-cutting aspects are modeled as use case parameters, which are mapping to alternatives? For BDD, these corresponds to a VM feature having one or more BDD features linked to it, or a feature having one or more scenarios linked to it, or a feature having one or more steps linked to it.

[Ghanam] recommends that only leaf nodes are linked to EATs. A leaf node can have multiple EAT nodes. Each EAT node links to one test artifact. Ghanam recommends the test artifact of choice to be the test page (similar to the Feature level of BDD.) So we’re basically saying that a feature can have multiple test artifacts linked to it. Ghanam doesn’t really say how to actually make the test artifacts variability aware.
Todo

Diagrams of how they map
Categorising ways to make BDD variability-aware

This section introduces some ideas and terminology related to variability implementation mechanisms, with a focus on requirements artifacts and test artifacts. With these terms we can categorise and evaluate different mechanisms.

The FOSD book describes three main dimensions of variability mechanisms: binding time, representation, and technology, and quality criteria that can be used to evaluate a variability mechanism.

7.1 Variability implementation dimensions

7.1.1 Representation

There are different ways to represent variability within an artefact. The two approaches used widely in practice are annotation-based and composition-based.

**Annotation-based approach**

In the annotation-based approach, artefacts are marked up in some way with annotations which determine what feature that part of the artefact belongs to.

A very common example of this is code implementations is the use of preprocessor directives such as #ifdefs.

In the context of Gherkin, tags could potentially be used for annotation.

**Composition-based approach**

In the composition-based approach, the parts of an artefact corresponding to a particular feature are contained within a dedicated file, container or module.

In code, there are various ways to achieve a composition-based approach, such as design patterns, frameworks and plugins.

7.1.2 Technology

The two main types of technologies available for variability implementations are language-based versus tool-based.
7.1.3 Binding times

The binding time is at which point the mechanism for distinguishing between variants of a feature come into play. The binding time might be compile-time, load-time or run-time.

An example of compile-time binding would be preprocessor directives, that conditionally include or exclude code at the point of compiling the application.

An example of a load-time binding implementation mechanism would be conditional execution of code via parameterisation, where the selected value of the parameter is loading from a configuration file.

Run-time binding can take into account real-time changes in the context of the application.

7.2 Quality Criteria

Quality criteria give us a framework to discuss different ways of making artifacts variability-aware. There’s no silver bullet approach that fits all situations, so it’s good to have a way to talk about the different options.


7.2.1 Preplanning effort

Preplanning effort is how much in advance you have to think about variability concerns. This is especially relevant in an agile context (reactive in SPLE terms), where we don’t want too much of a big-design up front domain engineering phase. For our first customer, we’re unlikely to want to put too much time into variability points for other customers. We’d like to be able to incorporate that variability in, as and when it comes, relatively easily, without much reengineering. In BDD terms, we would like to avoid having to do too much work with our first customer in order to add variability to the feature specs when it arises.

7.2.2 Feature traceability

Feature traceability describes how easily we can figure out what software artifacts contribute towards a given feature. If the code for a feature is scattered across many files, it makes it harder to reason about what code is implementing that feature. Likewise for other artifacts. In BDD terms, its unlikely we would get too much scattering across feature specs for a feature on the variability model.

7.2.3 Separation of concerns

In the context of variability, separation of concerns refers to how cohesive artifacts are in terms of which feature they relate to. If say a code file is full of references to lots of different features, it makes it more difficult to reason about, and to safely make changes to that file just a given feature. We can also say that the artifact is tangled. In BDD, it is unlikely that a feature spec will contain parts related to multiple features (although it is possible with e.g. crosscutting concerns.)

Todo

Reference bonafacia here.
7.2.4 Information hiding

7.2.5 Granularity

Granularity refers to the structural level at which the variability mechanism can efficiently reuse parts of an artifact. For example, in code, maybe the mechanism can conditionally include/exclude/replace at the class-level, or at the method-level. Or if the mechanism is very fine-grained it might be able to conditionally include/exclude/replace at the statement level. In terms of BDD we can think of supporting include/exclude/replace of a full feature spec file, or at the scenario level, or at the scenario step level.

Another way of thinking about granularity is as to how well the mechanism supports different types of variability: variability in function, variability in control flow, and variability in data [Bachmann]. It’s important to note that we can always support variability through duplication and amendment, but what we want to achieve is variability through efficient reuse.

7.2.6 Uniformity

Uniformity refers to how well the variability mechanism works across all artifacts in the lifecycle. If we use one mechanism for requirements, another for code, another for tests, this is going to be difficult to maintain.

As this relates to BDD, we investigate mechanisms that have been defined for code implementation and see how they apply to specs and tests.

The SPLE book [Pohl2005] provides some other quality criteria within its section on product line testing strategies.

7.2.7 Learning effort

How easy is it to actually understand and use the mechanism? For example, tool support helps here.

Todo

more on this

7.2.8 Overhead

Are they any unnecessary overheads associated with the mechanism that we might wish to avoid.

Todo

more on this

Todo

weighting of the importance of the criteria with regards to agile and BDD
8.1 Gherkin

As discussed in the section on Gherkin, Gherkin is a business-readable DSL for specifying system behaviour.

An example of feature specifications in a Gherkin feature file looks as follows (excellent ninja example from http://pythonhosted.org/behave/tutorial.html#feature-files):

```
Feature: Fight or flight

In order to increase the ninja survival rate,
As a ninja commander
I want my ninjas to decide whether to take on an opponent based on their skill levels

Scenario: Weaker opponent
  Given the ninja has a third level black-belt
  When attacked by a samurai
  Then the ninja should engage the opponent

Scenario: Stronger opponent
  Given the ninja has a third level black-belt
  When attacked by Chuck Norris
  Then the ninja should run for his life
```

When introducing variability into Gherkin, we can make use of some of the variability concepts described in Categorising ways to make BDD variability-aware.

- How do we represent the variability within the Gherkin file?
- At what level of granularity do we allow for variability?
- Can we have abstract features with specification tests, on concrete features only?

8.1.1 Representation

An annotation-based approach involves marking up the Gherkin file in some way with the feature the parts of the Gherkin file is associated with. It also requires a way of ensuring that the correct parts of the Gherkin specifications are executed when the feature corresponding to a particular annotation are selected in a product configuration.
8.1.2 Granularity

Granularity in the context of Gherkin refers to at what level elements of the Gherkin file we might wish to override or combine based on feature selections.

A course-grained granularity would mean we simply combine together full feature files based on which features have been selected.

A finer-grained granularity would allow for the combination of variant scenarios within an individual feature file.

An even finer-grain would allow the composition of individual scenario steps.

The most fine-grained level would be the arbitrary composition of any piece of a feature file, including parts of the individual scenario step descriptions. In practice this would be difficult.

Todo

give use cases/advice for when different levels of granularity might be required.

8.2 Step definition code

8.3 Progress reports

8.3.1 Pickles
Variability-awareness techniques for BDD

In this section we discuss ways in which the components of BDD can be made variability-aware. There are a variety of ways to do it, with no silver bullet that fits for all circumstances and contexts. For example, while one approach may be more sophisticated, it may simply have too much of a learning overhead for a small project. Hence we categorise and evaluate each approach against the quality criteria to enable informed decisions to be made.

With all approaches we make the assumption that we wish to avoid, as much as we can, overloading existing Gherkin language constructs. The existing keywords and structures should keep the same function as they have for single-system engineering, and we should add variability-awareness to Gherkin without confusing what these single-system concepts mean. The exception to this is the tags concept – which is a very open and general-purpose concept in Gherkin, so nothing is lost or confused by using tags for variability purposes.

We refer back to the quality criteria discussed earlier.

In the interests of the uniformity criteria, we take inspiration from the various mechanisms defined for code implementation, and see how they may apply to specs and to test artifacts.

9.1 Build system

Using the build system approach, each feature in the feature model can have one associated feature file. When a product is configured to include that feature, then the specification tests for that feature are included by the build system copying that file into the correct directory from which the feature files are run.

9.1.1 Implementation

We need a way of linking the feature to the feature file. And we need part of the build system to be able to copy the files to the correct location based on this link.

The simplest link mechanism is having the name of the feature file match exactly the name of the feature as gets included in the configuration file produced by the product configurator. Then the build script needs to just search for a file with that name and output it to the correct directory.

Todo

show an example in FAKE
9.1.2 Considerations

The granularity of this approach is at the feature level.

Todo

More considerations here

Todo

Does every feature have a feature file? Only concrete features perhaps?

9.2 Annotation by product

A simple annotation-based approach is to tag each feature file or scenario with the name of the customer(s) to which it applies.

Feature: Add Todo

@customer1 @customer2

Scenario: User adds valid todo
  Given the user has input the label 'Take over the world'
  When the user submits the todo
  Then the todo should be added to the user's todolist

@customer2

Scenario: User adds a todo with a description
  Given the user has input the label 'Take over the world'
  And the user has input the description 'Same thing we do every night'
  When the user submits the todo
  Then the todo should be added to the user's todolist

@customer1 @customer2

Scenario: User adds invalid todo
  Given the user has left the label blank
  When the user submits the todo
  Then the user should be prompted to add a label

This simplifies the running of the tests from the build scripts. We only need to pass the product/customer name as a filter to the tests, and the test runner will run only the scenarios that apply to this product/customer.

With this approach, all variability is contained implicitly within the feature files. There is no explicit representation of variability in a variability model. Any configuration of the product line for a particular customer means going through each feature file and tagging the features and scenarios that apply for the new customer.

Note that as tagging an entire feature means that all scenarios are also tagged with the same tag, we must only tag at the scenario level.

At the Gherkin-level, variability by control flow and variability by data can only be achieved by duplicating scenarios. At the step definition level, step definitions are reusable between scenarios through parameterisation.

c.f. PLUC
9.2.1 Categorisation

- Annotation-based
- Language-based

9.2.2 Quality criteria

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Rating</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preplanning effort</td>
<td>Good</td>
<td>We don’t need to think much about variability in advance. If a new feature comes along, we tag it with all the customers that it applies to.</td>
</tr>
<tr>
<td>Feature traceability</td>
<td>Poor</td>
<td>If scenarios from multiple feature files are related to one particular feature in the feature model, there is no simple way of determining that they all relate to one feature.</td>
</tr>
<tr>
<td>Separation of concerns</td>
<td>Poor</td>
<td>Feature files contain all variants. Variability concerns are hardcoded into the feature files.</td>
</tr>
<tr>
<td>Information hiding</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Granularity/variability-aware</td>
<td>Poor</td>
<td>We can only select at the scenario-level. Only variability by function is possible without duplication.</td>
</tr>
<tr>
<td>Uniformity</td>
<td>Poor</td>
<td>There is no effect on step definition code. It is all included, and selected for by the selection of scenarios at the Gherkin-level.</td>
</tr>
<tr>
<td>Learning effort</td>
<td>Good</td>
<td>No changes to Gherkin. Simply requires tagging scenarios with the customers to which they apply.</td>
</tr>
<tr>
<td>Overhead</td>
<td>Poor</td>
<td>There is duplication of scenarios. When a new product/customer is added, we have to manually amend all the feature files to tag the scenarios we should be including in their requirements and their tests.</td>
</tr>
</tbody>
</table>

9.3 Annotation by feature

An alternative annotation-based approach is to tag features/scenarios with the corresponding name of the feature from the variability model.

@add_todo
Feature: Add Todo

Scenario: User adds valid todo
  Given the user has input the label 'Take over the world'
  When the user submits the todo
  Then the todo should be added to the user's todolist

@todo_description
Scenario: User adds a todo with a description
  Given the user has input the label 'Take over the world'
  And the user has input the description 'Same thing we do every night'
  When the user submits the todo
  Then the todo should be added to the user's todolist

Scenario: User adds invalid todo
  Given the user has left the label blank
  When the user submits the todo
  Then the user should prompted to add a label
Variability itself has now been moved away from the feature files themselves, and can be modelled and configured using a dedicated feature modelling tool.

In order to run the tests for a particular product configuration, we filter the test runner by the names of the selected features from the feature model.

Note that without introducing new constructs and corresponding tooling, the tagging approach only affords us variability-awareness for variability by function.

Todo

Discuss Scenario Outlines with respect to variability by data

It would be possible to do so allow variability by control flow, or variability by data with new tag constructs and amendments to Gherkin. For example:

<table>
<thead>
<tr>
<th>Scenario: User adds valid todo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given the user has input the label '&lt;&lt;&lt;LABEL&gt;&gt;&gt;'</td>
</tr>
<tr>
<td>@todo_description</td>
</tr>
<tr>
<td>And the user has input the description '&lt;&lt;&lt;DESCRIPTION&gt;&gt;&gt;'</td>
</tr>
<tr>
<td>When the user submits the todo</td>
</tr>
<tr>
<td>Then the todo should be added to the user's todolist</td>
</tr>
</tbody>
</table>

But this would require introducing new ways in which to mark up Gherkin files which is something we would like to avoid, where possible.

c.f. PLUSS [Eriksson2005]

### 9.3.1 Quality criteria

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Rating</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preplanning effort</td>
<td>Good</td>
<td></td>
</tr>
<tr>
<td>Feature traceability</td>
<td>Poor</td>
<td>Feature files contain all variants. Variability concerns are hardcoded into the feature files.</td>
</tr>
<tr>
<td>Separation of concerns</td>
<td>Poor</td>
<td></td>
</tr>
<tr>
<td>Information hiding</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Granularity/variability-awareness</td>
<td>Poor</td>
<td>We can only select at the scenario-level. Only variability by function is possible without duplication.</td>
</tr>
<tr>
<td>Uniformity</td>
<td>Poor</td>
<td>There is no effect on step definition code. It is all included, and selected for by the selection of scenarios at the Gherkin-level.</td>
</tr>
<tr>
<td>Learning effort</td>
<td>Good</td>
<td>No changes to Gherkin. Simply requires tagging scenarios with the customers to which they apply.</td>
</tr>
<tr>
<td>Overhead</td>
<td>Neutral</td>
<td>There is duplication of scenarios.</td>
</tr>
</tbody>
</table>

### 9.4 Feature module composition

FeatureHouse is a tool that can be used for the composition of various types of software artifacts. It allows for the artifacts to be structured in feature modules as part of feature-oriented software development.

“FeatureHouse is a general approach to the composition of software artifacts. FeatureHouse is language-independent in that software artifacts written in various languages can be composed, e.g., source code,
test cases, models, documentation, and makefiles. Software artifacts are represented as feature structure trees (FSTs), which capture the essence of an artifact’s modular structure in the form of a tree. As composition paradigm, FeatureHouse supports two approaches: superimposition and three-way merge. FeatureHouse consists of a tool for superimposition, called FSTComposer, a tool for three-way merge, called FSTMerge, and a plug-in and generation mechanism for integrating new languages automatically, called FSTGenerator.

—http://fosd.net/fh

To enable Gherkin for usage within FeatureHouse we are implementing a feature structure tree grammar for Gherkin files.

Listing 9.1: AddTodo.feature in AddTodo feature module

**Feature:** Add Todo

**Scenario:** User adds valid todo

  *Given* the user has input the label 'Take over the world'
  *When* the user submits the todo
  *Then* the todo should be added to the user's todolist

**Scenario:** User adds invalid todo

  *Given* the user has left the label blank
  *When* the user submits the todo
  *Then* the user should prompted to add a label

Listing 9.2: AddTodo.feature in TodoDescription feature module

**Feature:** Add Todo

**Scenario:** User adds a todo with a description

  *Given* the user has input the label 'Take over the world'
  *And* the user has input the description 'Same thing we do every night'
  *When* the user submits the todo
  *Then* the todo should be added to the user's todolist

**Todo**

Add diagrams of how the feature modules would look like

Thus, if a product configuration includes the *AddTodo* and *TodoDescription* features, FeatureHouse will merge these two feature files into one that contains the scenarios to be tested against that product configuration.

In terms of running the tests, the product-specific feature files are output into one directory for that specific product. The test runner just needs to be pointed at this directory to find the tests for the customer. No tag filtering is required.

As it stands, this only makes the feature files variability-aware in the sense of variability by function. Variability by control flow or variability by data can be achieved by duplication of test artifacts.

**9.5 Modeling scenario variability as crosscutting mechanisms**

c.f. MSVCM [Bonifacio2009].

The aspect-oriented approach provides a compositional approach with which to make the feature files variability-aware in terms of variability by control flow.
It enables adding arbitrary scenario steps before and after existing scenario steps.

**Todo**

Describe differences between homogenous and heterogenous crosscutting concerns.

### 9.6 Aspectual feature modules

A combination of the feature module and aspect-oriented approaches.

The combination allows variability-awareness in terms of variability by function, variability by control flow, and variability by data.

### 9.7 Summary

The table below summarises different approaches to implementing variability in Gherkin based on various quality criteria.

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Build system</th>
<th>Gherkin tags</th>
<th>FeatureHouse</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Granularity</strong></td>
<td>Feature</td>
<td>Feature, scenario</td>
<td>Feature, scenario, scenario step</td>
</tr>
<tr>
<td><strong>Learning curve</strong></td>
<td>Simple</td>
<td>Simple</td>
<td>Understanding feature module packaging mechanism</td>
</tr>
</tbody>
</table>

As far as ApleT goes, we would recommend starting out with something that has a low preplanning effort, low overhead, and low learning effort. Then refactoring towards more advanced techniques.

We would recommend avoiding clone-and-own by version control unless you are already doing it, or it is desirable for some other reason (organisational issues say). We’d recommend starting with annotation by feature, although this does require somewhere recording feature configurations for each customers (this doesn’t necessarily have to be through a feature modelling tool, however, it could simply be manually maintained text files.)

**References**
ApleT is a framework that helps you to create software product lines using behaviour-driven development. ApleT builds on the concepts of variability-awareness in behaviour-driven development artifacts.

The ApleT process describes the stages involved in order to do this, including the roles, activities, and input and output artifacts of each stage.

ApleT is designed to be used in various organisational contexts and different software stacks. The ApleT toolchain outlines a general toolchain required and gives examples of what provides these different tools in different stacks.
The ApleT framework includes a process that should be followed during the production or maintenance of products in the product line. The trigger for a new cycle through the evolution process is a change in requirements – this is either through the development of the software for a new customer, or the amendment of the software for an existing customer.

0. **Entry Point.** The entry point to the process is a suite of products that are covered by executable specifications (variability-enabled or not.) Each product has a variability profile and a build configuration, and the executable specifications are run as part of a continuous build system process that consistently outputs deployable software.

1. **Variability Evolution.** When new variability is required, whether by a new product in the product line or a new feature for one or more products in the product line, requirements are gathered and the variability model is generalised. This is captured in the FeatureIDE modelling tool.
2. **New Variability Profile and Build Configuration.** New variability profiles are produced. In the case of a new product, a new variability profile is produced for the whole product, and a new build configuration is introduced for the build system. The variability profile is created in the FeatureIDE modelling tool, and the job build configuration is constructed in Jenkins and FAKE.

3. **Naive Product Instantiation.** A naive product instantiation is enabled in the product line implementation, which using the new profile and build configuration the build system can build, deploy and run executable specifications against. This naive product instantiation may be a complete source clone, or hopefully it will be a case of configuring a more advanced product line implementation.

4. **Implement new variability.** For any new variants introduced, BDD specifications are generated and implemented that encompass the new variability. This is done iteratively, feature by feature.

   a. Create (Failing) Executable Specification. For the new variant, create a new executable specification.

   b. Implement specification. Dropping down into the low-level red-green refactor cycle until the specification acceptance test is passing.

      (a) Repeat for all new variability.

3. **Refactor.** When all specifications are passing for all products, refactor the product line implementation.

4. **Repeat.**

Throughout this cycle, test reports are generated by the build system process to provide feedback on the state of all products in the product line.

---

**Todo**
fit the ApleT process into the different quadrants defined in sple process
Chapter 11. The ApleT Process
The ApleT Toolchain

The ApleT process defines a number of tools that are required.

- Feature Modelling Tool. Used to produce feature models.
- Product Configurator. Used to configure instances of products for individual customers.
- Feature File Composer. Used to compose together variable Gherkin feature files into specific versions for each configured product.
- CI Server. Used to build, test and deliver products and reports.
- Product Builder. Used to build products as part of a continuous integration process.
- Spec Runner. Used to run the specification tests for the products.
- Report Generator. Used to produce real-time progress reports that provide feedback on the status of product feature implementations.

The idea is that the ApleT process should be reusable across different development stacks, be it .NET, Java, Ruby, Python, Node, or some combination of these.

### 12.1 Reference implementation of ApleT in .NET

- Feature Modelling Tool = Feature Model DSL.
- Product Configurator = Feature Model DSL.
- Feature File Composer = FeatureHouse.
- Product Builder = FAKE.
- CI Server = Jenkins.
- Spec Runner = SpecFlow.
- Report Generator = Pickles.
12.2 Example Java toolchain for ApleT
ApleT Roles

Todo
The roles required in the ApleT process.

- SPL Owner
- Feature Modeller
- Product Owners
- VFF Producer
- Build Scriptor
• SPL Developer
This section will show a worked example of creating and evolving a software product line using ApleT. This will go through all of the steps within the ApleT process, using the tools from the .NET ApleT toolchain.

We will work through a very simple web app product line, the TodoMonkey product line for todo list applications as introduced by Schnapptack.

### 14.1 The TodoMonkey product line

The feature model of the full TodoMonkey Product Line looks like this:

![Feature Model Diagram](image)

**Todo**

This feature model might need some refining, e.g. break base into list, add item, label, mark as done, etc.

Feature base is mandatory (filled circle). All other features are optional (empty circle). However, feature search depends on feature filter.
**base** provides a basic todo list. Items can be added. Items have a label and can be marked as done.

**description** adds a description field to each item. Double click items so see the field.

**filter** adds filter functionality: show all, show done, show not done.

**search** adds a search field (searches all the labels).

In this example, we will add a bit of history to the TodoMonkey product line, and show how it could have evolved over time.

### 14.2 The process

#### 14.2.1 Feature model

First we will create a feature model. To begin with, we are making this for just one customer, and they only want the most simple, base implementation. The feature model is created and amended by the Feature Modeller using the Feature Modelling Tool. In this case, that’s us, and we’ll use the Feature Model DSL Visual Studio extension.

![Feature Model Diagram](image)

#### 14.2.2 Product configuration

Customer A wants all the features. After all, that’s why we’re building the product. Let’s make a configuration for Customer A.

#### 14.2.3 Specifications

Let’s make a specification for the Base feature. We’ll put it in a folder called *Specifications* for now.

<table>
<thead>
<tr>
<th>Feature: Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>In order to keep track of things I need to do</td>
</tr>
<tr>
<td>As a Todo Monkey</td>
</tr>
<tr>
<td>I want to be able to maintain a list of todo items</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scenario: Add a todo item</th>
</tr>
</thead>
<tbody>
<tr>
<td>When I create a new todo item named 'Buy milk'</td>
</tr>
<tr>
<td>Then the new todo item should be in my todolist</td>
</tr>
</tbody>
</table>
Scenario: Mark item as done
   Given I have the following todolist:
     | Todo    |  
     | Buy milk |  
   When I mark the 'Buy milk' item as done
   Then the 'Buy milk' todo should be recorded as done in my list

Feature: Add Todo

   In order to keep track of the things I need to do
   As a TodoMonkey
   I want to add todo items to a todolist

Scenario: User adds valid todo
   Given the user has input the label 'Take over the world'
   When the user submits the todo
   Then the todo should be added to the user's todolist

Scenario: User adds invalid todo
   Given the user has left the label blank
   When the user submits the todo
   Then the user should be prompted to add a label

14.2.4 Build a product and get it tested

Todo

talk about jenkins config, e.g. we need git and msbuild plugins

Todo

need to reference the GOOS book more for walking skeleton, outside in ideas, etc.

Now we need to add a build configuration to our continuous integration setup, that will 'continuously' build the product for Customer A and run the specification tests against it so we and the product owner can see how much we have done.

See walking skeleton.

So first of all we need a new build profile in Jenkins.
We also need a very naive product implementation. Something that we can run the specification tests against. We’ll create a basic web app.

We also need to create a build script that Jenkins can run and produce and deploy our naive product, in order to run tests on it.

The script could be some combination of Jenkins’ own build steps, and calling out to a build scripting language such as FAKE.

Todo
write the test execution code, here probably use browser automation

```csharp
[Given("the user has input the label '.+\')")]
public void GivenTheUserHasInputTheLabel(string label)
{
    ScenarioContext.Current.Set(label, "TodoLabel");
}

[When("the user submits the todo")]
public void WhenTheUserSubmitsTheTodo()
{
    var label = ScenarioContext.Current.Get("TodoLabel");
    var result = todoService.AddTodo(label);
}

[Then("the todo should be added to the user's todolist")]
public void ThenTheTodoShouldBeAddedToTheUsersTodolist()
{
    result.ActionSuccessful.Should().BeTrue();
}
```

14.2.5 Progress report

**Todo**

discuss manual vs automated build runs in jenkins

We are now able to run the build within Jenkins, which produces a progress report for us.

**TodoMonkey Test Execution Report**


<table>
<thead>
<tr>
<th>Summary</th>
<th>Features</th>
<th>Success rate</th>
<th>Scenarios</th>
<th>Success</th>
<th>Failed</th>
<th>Pending</th>
<th>Ignored</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 features</td>
<td>0%</td>
<td></td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature Summary</th>
<th>Feature</th>
<th>Success rate</th>
<th>Scenarios</th>
<th>Success</th>
<th>Failed</th>
<th>Pending</th>
<th>Ignored</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>0%</td>
<td></td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature Execution Details</th>
<th>Feature: Base</th>
<th>Scenario</th>
<th>Status</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add a todo item</td>
<td></td>
<td>pending</td>
<td>0.188</td>
<td></td>
</tr>
<tr>
<td>Mark item as done</td>
<td></td>
<td>pending</td>
<td>0.013</td>
<td></td>
</tr>
</tbody>
</table>

As we can see, the status of the specification tests is ‘pending’ - we haven’t written any test layer code yet.

14.2.6 Writing the test layer code

We want to write the test layer code that tests our production system.

We have a lot of freedom in how we write this test code – it could be full end-to-end tests, where we exercise the entire system from the interface by automation, or it could be a lower level integration test that might avoid the interface. Interface automation has its benefits and its pitfalls – one main benefit being we know we are exercising the system
as a user would. However if not carefully written, automation tests can be brittle. And they can be slow to execute, which leads to issues of scale.

Todo

more discussion of automation testing vs lower level

We're going to write tests that bypass the interface for now. We're going to test our first scenario.

<table>
<thead>
<tr>
<th>When</th>
<th>I create a new todo item named 'Buy milk'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Then</td>
<td>the new todo item should be in my todolist</td>
</tr>
</tbody>
</table>
Multi-tenancy

Multi-tenancy is a common pattern in web application development, for serving up to multiple “tenants” customised versions of the same application.

All of the work on software product lines implementation applies to multi-tenancy. Most multi-tenant sites can be viewed as software product lines, with the individual product being defined at load-time.

15.1 Resources

- Building Multi-Tenant Applications with Paul Stovell is a really good discussion of the operations side of product lines.
- The Force.com Multitenant Architecture
The Screaming Architecture and Obvious Architecture ideas are quite similar to feature-oriented software development.

- http://developer.7digital.com/blog/obvious-architecture
This page is currently a general dumping ground for various ways of implementing variability/feature-oriented development when it comes to .NET web applications.

17.1 Blog Posts

The comments of these posts are always worth a read for pros and cons.

- ASP.NET Core - Feature Slices for ASP.NET Core MVC
  Steve Smith outlines the steps needed to get a basic feature folder setup in ASP.NET Core MVC. He discusses how it can be achieved both with the built-in Areas functionality, or by rolling it yourself with an IControllerModelConvention and an IViewLocationExpander. The main differences between the two are: Areas is built-in and requires minimal effort to get it working, however it’s still quite ‘folder-heavy’. The custom Feature Folders approach require more effort to get working, however all artifacts are contained within one folder for each feature that is easier to navigate. Steve shows a way to combine both approaches, so you can have feature folders underneath Areas. This seems like a good way to keep Areas what they are for (separating out large distinct areas of an application), and still being able to have the parts of the Areas navigable by feature names rather than technologies.

- http://marisks.net/2016/02/16/feature-folders-vs-tech-folders/
- http://stackoverflow.com/a/1528571
- https://github.com/achingono/Enfold
17.2 What needs composing in an ASP.NET MVC web application?

- Controllers
- Routing
- View Models
- View Templates
- JavaScript
- CSS
- Static files
- web.config

17.2.1 Controllers

Controllers are C# code so can make use of the standard mechanisms - #ifdefs, parameterisation, design patterns. If we were to use feature-oriented composition FeatureHouse C# composition could potentially be used.

17.2.2 Routing

Action methods on controllers can be attributed with routes, and the routing engine picks them up from there, so it could be that composing controllers takes care of composing routing.

17.2.3 View Models

As for controllers.

17.2.4 View Templates

One simple approach is extended ViewEngine that allows for view overriding. By making use of partial views this allows overriding at quite a fine level of granularity.

Note that the folder structure here has it broken down by customer/product. For branding concerns, we have to really think of branding as a feature variant that only applies to one customer. So where the alternate views are for branding concerns, it’s OK to have them separated out essentially ‘by product’.

In the case of views being required only for a particular functional feature, we might have a single view engine that searches for views in feature modules, or we could even experiment with a view engine being bundled with every feature and added to the collection of ViewEngines at application startup, based on product configuration.

17.2.5 JavaScript

https://github.com/henzk/featuremonkey.js

17.2.6 CSS

CSS is composable by simple concatenation.

17.2.7 Static files

...

17.2.8 web.config

Web.config transforms in Visual Studio can deal with some composition of configuration options. It’s limited though in that you can only have overriding configs per build configuration and per publish profile. There’s no real way to separate this out by feature.

17.3 ASP.NET MVC Areas

Areas provide a means for separating an application up into distinct areas. It’s intended more for large subsections of an application however, so not at the fine-grained level of features.

17.4 ASP.NET MVC Portable Areas

https://lostechies.com/erichexter/2012/11/26/portable-areas-3-years-later/

17.5 FubuMVC bottles

FubuMVC’s bottles were a way to package up code, web endpoints and views in a single place that could be reused in web applications.

I never worked with it, and I don’t know how it went about composing different artefacts (it seems to be runtime composition?), but it seems to have definite similarities with the idea of feature modules in FOSD.

https://lostechies.com/josharnold/2011/09/05/modularity-via-bottles/

17.6 OrchardCMS

Orchard builds on top of ASP.NET MVC and has the idea of extensibility via modules built into it from the ground up. Behind the scenes, Orchard modules are built on top of the ASP.NET MVC Areas mechanism.

http://docs.orchardproject.net/en/latest/Documentation/How-Orchard-works/
CHAPTER 18

Other web application product line frameworks

18.1 Python/Django

- https://github.com/henzk/django-productline

18.2 Ruby on Rails


18.3 PHP/Symfony

CHAPTER 19

References

References
Todo

feature traceability

Todo

This feature model might need some refining, e.g. break base into list, add item, label, mark as done, etc

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/plet/checkouts/latest/docs/plet-example.rst, line 21.)

Todo

talk about jenkins config, e.g. we need git and msbuild plugins

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/plet/checkouts/latest/docs/plet-example.rst, line 107.)

Todo

need to reference the GOOS book more for walking skeleton, outside in ideas, etc.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/plet/checkouts/latest/docs/plet-example.rst, line 108.)

Todo

write the test execution code, here probably use browser automation

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/plet/checkouts/latest/docs/plet-example.rst, line 134.)

Todo

discuss manual vs automated build runs in jenkins
Todo

more discussion of automation testing vs lower level

Todo

fit the ApleT process into the different quadrants defined in sple process

Todo

The roles required in the ApleT process.

Todo

image of progress report

Todo

pickles

Todo

variability model

Todo

artifacts dependencies
Todo  
feature traceability

Todo  
give use cases/advice for when different levels of granularity might be required.

Todo  
Reference bonafacia here.

Todo  
more on this

Todo  
more on this

Todo  
weighting of the importance of the criteria with regards to agile and BDD

Todo  
show an example in FAKE
Todo
More considerations here

Todo
Does every feature have a feature file? Only concrete features perhaps?

Todo
Discuss Scenario Outlines with respect to variability by data

Todo
Add diagrams of how the feature modules would look like

Todo
Describe differences between homogenous and heterogenous crosscutting concerns.

Todo
Diagrams of how they map
• **I don’t think feature model features have to correspond directly to BDD features.**

  – At least if we’re using feature modules. A feature module for a feature could contain multiple feature files.

  – Might not be as easy for the other implementation methods.

• Need to read up a little of features and scenarios - how logically related should scenarios be?
Indices and tables

- genindex
- modindex
- search


