UML meets XP

Position statement for UML + extremity workshop, UML2001

Alan Cameron Wills

Trireme International Ltd
www.trireme.com *alan@trireme.com*

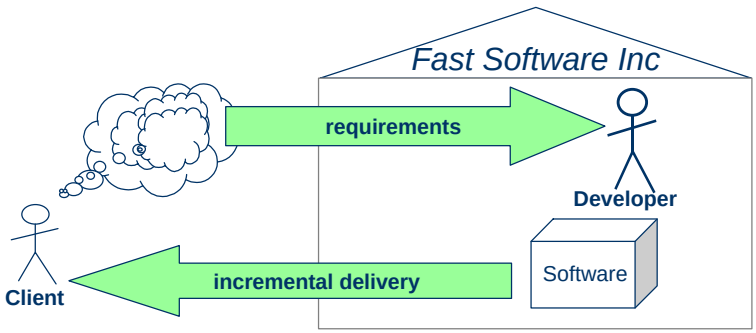
I have always been a strong believer and practitioner of the JFDI approach to development, which yields high morale, early identification of problems in design and of any misunderstandings of the requirements, and easily tracks changing needs as they occur. I applaud eXtreme Programming [Beck] for enunciating some excellent guiding principles that separate this approach from iterative hacking.

And yet I also have always believed in using models to understand abstractions during the design; and to communicate them to others both during the design and afterwards. The one aspect of XP I'm not keen on is the disdain for documents and models other than those enshrined in the code and tests. Nevertheless, it's undeniable that any documents separate from code and tests can and will get out of step with them, and will not be read.

This position paper outlines some of my experience, as a consultant methodologist with a number of clients, in attempting to get the benefits of XP at the same time as those that follow from using modeling techniques.

What does XP give us?

XP is a rigorous approach to software development, in the sense that specifications are carefully encoded before implementation, and the latter is verified against the former: executable tests are arguably the most practical kind of specification. Unlike earlier rigorous methods (such as [Jones], [Morgan]) XP answers the question "but how do you know you've got the specification right?"



Tight customer-developer loop ensures that any misunderstandings in the developer's mind about what the customer wants are soon corrected. The same goes for any misunderstandings in the customer's mind about what is needed.

Regression testing makes it possible to elaborate and refactor the code in many small increments, without the bug count going out of control. XP developers reckon that up to 50% of their time is spent designing unit (and integration) tests; the investment pays off in robust and reliable resulting code – if the tests are designed well.

If we are to try to improve on orthodox XP, we should not lose these characteristics.

Why do we need more than XP?

Communication and discussion of abstractions. Most of us can't hold a design in the head without some means of writing down various abstractions: requirements, relationships, collaborations. UML is generally a better medium for this purpose than code, whether of implementations or tests. Automatic test code may be the most practical form of specification; but it is not very readable as a modeling medium. In my work with medium to large projects, we have found it impractical to use program code and informal stories as the only means of discussing the requirements and the architecture of the system – especially where there is a large team made up of people of varied experience.

Separation of analysis and development roles. Some people are better at talking to customers and warmly empathizing with their needs; others are better at neat, cool, performant solutions. We find it necessary to separate these roles.

Component interfaces, design patterns. Any kind of generalization needs (a) a process for abstracting and generalizing; and (b) a notation for describing how to use the result. In component based development, it is important to be able to describe the component interfaces, and the concepts with which they deal, separately from the components themselves.

Roles of UML

Many tools support a direct correspondence between UML and program code, and can be parameterized to work within different architectures and idioms. Designers can effectively write programs in UML.

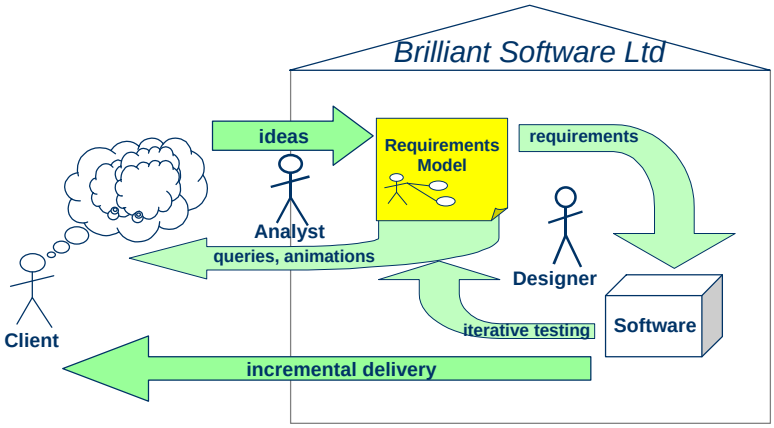
I find these features very useful, but not sufficient for the development job. There are important cases where I want to create models that do not represent the implementation:

- In component based development, it is important to write definitions of the interfaces or connectors, separately from the implementation of any component; and separately even from the specifications of any one component.
- The same is true in enterprise application integration: the diverse systems you are integrating must speak a common language at their interfaces, but may have very diverse and unknown implementations.
- Implementations typically have many classes that are not intrinsic to the problem, but are introduced in order to improve performance or maintainability. In analysis, I want to stick to describing the problem rather than the solution.

In our approach, we therefore separate two uses of UML: requirements models define the behavior visible from outside a system or component; implementation models are used by designers to define the internal structure of the system. Requirements models say nothing about the internal implementation.

Using a Requirements Models within an XP-like process

In consultation with our clients, we have gained some experience in meeting the above needs, whilst preserving the benefits of XP. Of course, the result has much in common with other iterative processes, such as RUP. However, we note a number of improvements on other approaches. For now, we'll call this approach XP/RM.



The process introduces a Requirements Model, in which is captured everything we think we are going to provide that will be visible to the customer. That document will typically include a UML model of the functional requirements; but also non-functional requirements (performance, robustness, usability, etc). It will include use-cases, classes, business rules and protocols. It captures everything the developer needs to know to create the software.

The Requirements Model is created and maintained by systems analysts, who may be separate from the developers. The developers treat the analyst as their customer. The Requirements Model is a specification of the required system behavior – not an implementation model that reflects the code. (Although the developers might use implementation models for their own purposes.)

Notice that there are now two major loops: the Analyst's loop, which incrementally builds a model of the customer's requirements; and the Designer's loop, which incrementally builds software to meet those Requirements.

It is important to stress that the arrows in the diagram represent flows of information that operate more or less continuously in peristaltic fashion. The designers do not wait until the model is anywhere near complete before beginning to create software. The model should be developed in parallel with, or in a slight lead of, the software. The designers may talk to the clients and should sit in on some of the modeling sessions; analysts should monitor development closely, and should represent the customer to the designers.

The Analyst's Loop

In XP, the loop between the customer and the developer ensures that the developer's successive approximations to the required solution are kept on track, and are responsive to changes in requirements that occur during development.

In XPRM, the loop between customer and analyst builds a model by successive approximation. Good analysts have always worked closely and continuously with their clients; but we emphasize:

- The need to work from broad sketch to more precise detail in key areas, so that the developers can start work. We usually start by identifying the principal concepts and business use-cases, together with rough non-functional requirements in the areas of response times, security, numbers of users, and storage requirements.
- Applying analysis techniques that tend to expose ambiguities and inconsistencies in the requirements as stated by the customer. Standard use-case analysis doesn't score highly in this respect; we use multiple overlapping views and lightweight cross-checks (detailed below). The purpose is to feed a stream of searching questions back to the client, resulting in more accurate model than would otherwise have been obtained.

Notice that the Requirements Model is placed in-house. We don't necessarily expect the analyst's model to be part of the customer contract. Its purpose is as a technical document for us to clarify and agree between ourselves what we're providing to the customer. It's part of the analyst's role to interpret the model to the customer, either by animation or storyboards, or by agreeing requirements statements, or just by asking questions while the model is being constructed.

The Designer's Loop

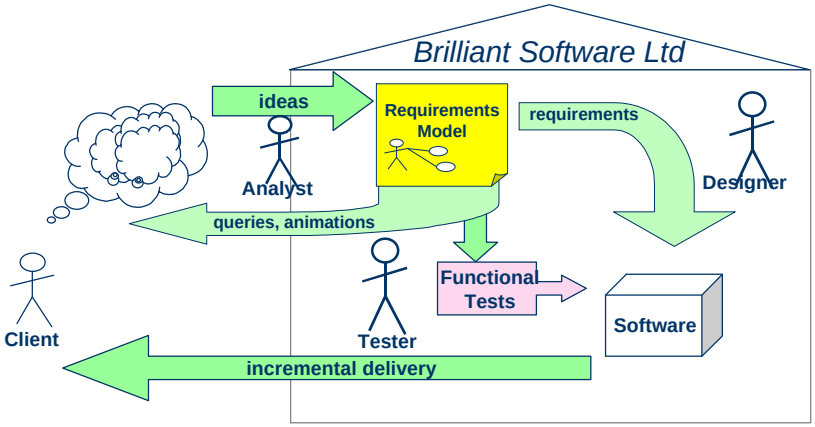
Designers work from the requirements, as captured in the Requirements Model. The designed software is tested against the model as it is developed; and is demonstrated to the customer at regular critical points (usually about every two weeks). The collusion between designers and customers is slightly less close than in XP: the effect is to funnel requirements through the analysts. This can be especially helpful if the customers are actually a diffuse group.

The designers may use UML to make an implementation model: that is, a model that directly reflects the structure of the program code. This kind of model is supported very well by tools such as Together, Rose, and some others. Code is generated directly from the diagrams, and vice versa.

The designers may use UML to represent the implementation; and they might or might not do the design by making simple extensions of the Requirements Model; but we don't focus on that here.

Testing

The process of testing against the requirements model is more accurately represented as a flow from model to test software. In a substantial project, there is a separate test team to make this happen. We should distinguish between *unit tests*, which are written by the developers themselves for internal structures; and *functional tests* which test the entire system or component, and are derived from the Requirements Model.



(This process model doesn't show any unit tests, just as it doesn't show any models that are internal to the design process.)

Benefits of upfront requirements models

Our experience to date is that this approach has a number of benefits:

- **Rapid exposure of gaps and inconsistencies**. Applying our checking techniques while creating a model raises searching questions about the stated requirements very early on: more rapidly than even a manic XPer could generate prototype code. Thus many misconceptions can be ironed out more rapidly than with pure XP.
- **High-level requirements view**. Putting the customer in front of a working prototype can sometimes focus the mind unhelpfully on the user interface. Asking the customer about use-case scenarios, goals of use-cases, relationships between objects, and business rules, tends to give more useful feedback.
- **Damping requirements swings**. Customers have a tendency to make contradictory statements about what they want, and may change the requirements and later change them back again. The cause may be different factions within the customer organization; or uncertainty in the business world. The interposition of an analyst between the customer and the developers tends to even out some of these wild swings.

But to make the combination of UML upfront work with XP, we have to consider:

- how to avoid jeopardizing XP's responsiveness to change;

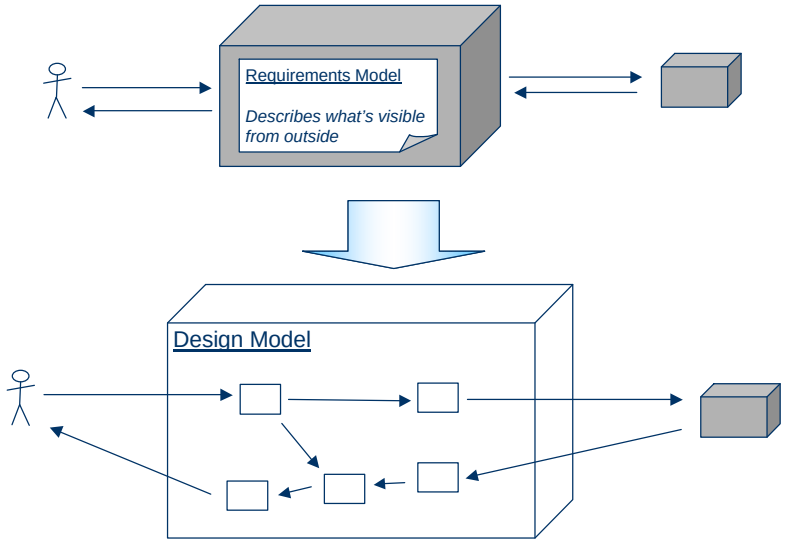
- how to converge on the correct requirements in the analysis loop;
- the relationship between use-cases and incremental development tasks;
- the relationship between tests and the requirements model.

Building Requirements Models in XPRM

What's in a Requirements Model?

The Requirements Model says nothing about internal structure: no responsibilities or operations are assigned to classes (unless they are visible at the external interface), and no collaborations are defined between objects. The only operations are those visible at the external interface to the system or component being specified; the only collaborations are those in which the component being specified takes part. A requirements model does define classes, but only those intrinsic to the problem domain.

The process of design decides how the implementation works and, if it is an OO or component based solution, decides what the objects or components shall be, what their individual responsibilities shall be, and how they collaborate to fulfill the Requirements.



A Requirements Model contains:

- Use cases, documented largely according to [Cockburn]; but with substantial emphasis on postconditions – the effect of each operation the system is asked to perform. This is particularly important at "sea level" (in Cockburn's terms: use-cases at the level of detail that describes individual transactions performed by the system or component, such as "place an order" or "accept stock delivery").
- Class diagrams declaring all the concepts used in the use-case postconditions. The classes do not have operations: behavior is described in the use-cases. (Assignment of responsibilities to classes is a designer's job.) The class diagram does not describe the internal structure of the implementation; it describes relationships between concepts understood by the system.
- Business rules, defined in terms of constraints on the concepts, and constraints on sequences of use-cases.
- Non-functional requirements, described in terms of the classes and use-cases.

Model elaboration techniques

Analysts construct models by using a set of techniques or patterns directed at creating a model in successive approximations, starting from something very simple, and elaborating in short steps. Some

of the steps are exploratory, asking the client directed questions about the business and the system required; others are checks designed to expose ambiguities or inconsistencies.

In our approach, multiple views are constructed, which must tie up with each other. There are multiple 'vertical' views – the system or component as seen by different users, through different interfaces: they must have the same meanings for common concepts, follow common rules where concerns overlap, and have compatible expectations of the system where they ask it to do the same thing; we won't go further into vertical views and their composition here.

There are also multiple 'aspects': these are different formalisms that abstract different kinds of information: for example, use-cases, class diagrams, and statecharts. We make these tie up in a way that is not currently widespread among analysts; and find it a very effective way of discovering ambiguities and inconsistencies.

We have a number of patterns for this purpose. Rather than reciting them directly, we'll illustrate just a few.

Spot nouns and verbs

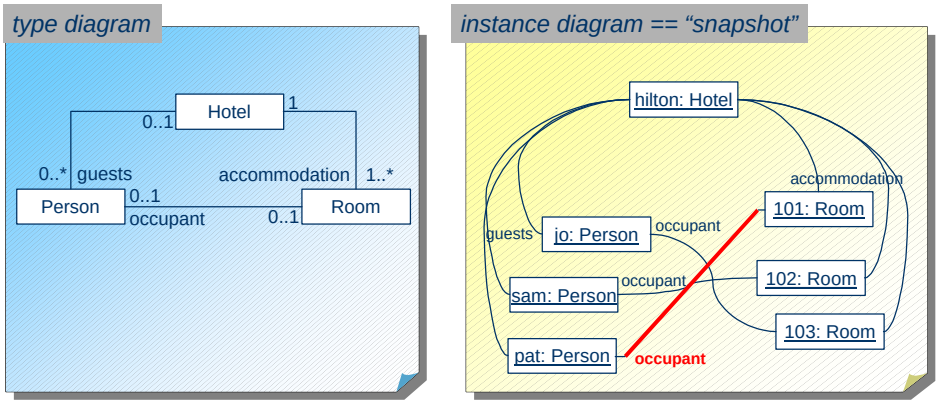
You've heard this one before. Nouns –> object types, verbs –> use-cases. We construct both kinds of view in parallel. This gives a first approximation to a model.

Use cases specified in terms of type model

For example, if we have:

> **use case** Check Guest into Hotel
> ***post:*** The Guest is now the occupant of a room in the Hotel, that was previously unoccupied.

We can envision the situation before and after an instance of this use-case in an instance diagram, by representing some instances of sample Rooms, Guests, and *occupant* relations between them. The straight red link (pat–101) represents the change to the "after" situation.



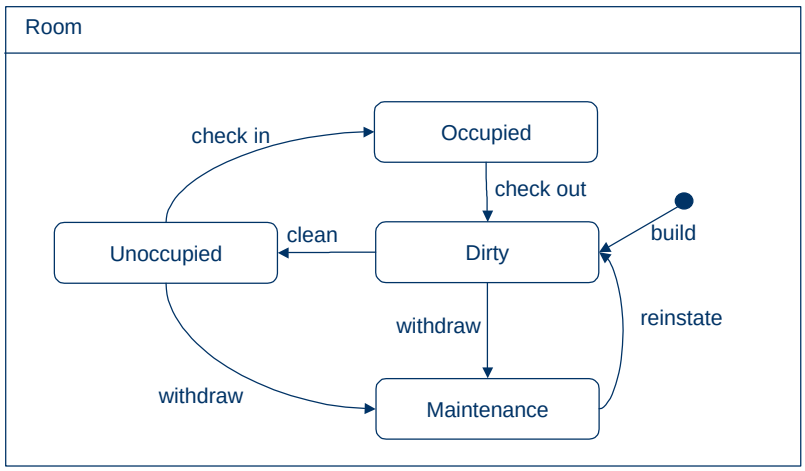
From all of the instance diagrams used to illustrate all the use-cases, we can derive a type diagram summarizing the concepts used in the use-case model, and which are sufficient to explain it.

An analyst might begin by asking about what happens in the hotel, document the use-cases, and from there derive a type diagram. Developing the two things works in parallel.

A number of standard questions can then be asked about the types and associations we've found: what are their cardinalities (how many Rooms can a Person occupy? etc); what else happens to these objects and associations? One way of investigating that question is to draw a statechart for a selected type.

Descriptive Statechart

By asking "how is one of these created?" and repeatedly "what might happen to it then?" we can create a statechart. (In a requirements model, these are purely descriptive: states are just Boolean assertions.) For example:



The transitions here are use-cases, which have been discovered by the process of drawing the statechart. We have also discovered some of their pre and postconditions – e.g. a precondition of *withdraw (a room from service)* is *Dirty or Unoccupied*. We can now write specifications for each of these use-cases.

Association types

Any association can be turned into an object, if it has useful attributes and states of its own. For example, we might create an *Occupancy* type to represent the *occupant* association, and then draw a statechart of it – thereby discovering the fate of cancelled bookings, unpaid stays, changes of room, etc.

Static constraints

Wherever there is a loop in the associations in the type diagram, there is a question about whether the instances must, must not, or might have an equivalent loop. Referring to the type diagram above: if a *Person* (e.g. Jo) is a *guest* in a *Hotel* (e.g. the Hilton), must the *Room* Jo occupies be part of the *accommodation* of that same *Hotel*? (The answer might be No, if the Hilton could have an arrangement with a neighbor for dealing with overbooking; it depends exactly what the associations signify: if the guest link points at the hotel you pay your bill to, while the accommodation link points to the Hotel the room is physically part of.)

Questions like this reveal certain kinds of business rule, that the system would have to observe.

Summary: Model elaboration patterns

Model elaboration techniques like these (and others – we have a battery of them) help to ensure that a broad, accurate view of the customer's requirements is obtained, without the undue focus on the user interface that arises from prototypes. This is a considerable improvement on plain XP.

The formalization of the requirements can be almost as effective as writing a prototype, in terms of ironing out ambiguities and gaps in the requirements; but it can be done by analysts who are expert in analysis rather than programming; and can be done separately for different views of the system, even if they will overlap in an implementation.

Component Based Development

eBusiness demands rapid response from the software development trade: and one of the ways we can achieve that is by building software from large or small components. This works best when the components form a kit of interchangeable parts – as in the Java windows system, or the Unix standard library. Each kit is designed to work well within a certain domain.

In such a component kit, we can make models of a number of things:

- The design of any one component. The model represents the classes in the program code, their responsibilities, and the collaborations between them.

- The specification of a component. Since there may be several interchangeable designs that meet the same specification – for example, to store sets of things, or to sort things – it is necessary that the specification model does not imply anything about the design.
- The specification of a connector. If we want that the components can be assembled in many configurations (like Lego), then we need to define a small number of standard interfaces. In Unix, the pipe is the standard connector; in Java Beans, there are events and properties; in Java AWT, there is the window-containment relationship, and in Java I/O, the filestream. The fewer connectors in the kit architecture, the more flexible the configurations available. These connectors have to be carefully defined, especially if third parties are going to create components for the kit.

In the last two cases, the model has no connection with any design, except that a design should conform to the behavior it stipulates.

If the components are developed according to the principles of XP, the specifications should be rendered as tests. There is an argument that the tests are the best specification, and that diagrams etc are superfluous. But I find that tests obscure the overall picture: they are difficult to abstract from and manipulate in the mind. Therefore, I would prefer to write component and connector specifications in UML/OCL/natural language; and derive the tests from those models.

It is clear that in the context of CBD, it is by no means the whole story to use UML as a programming language. The essence of pluggability is specification, purely of externally-visible behavior; and the ability to check conformance to specification.

Deriving Tests from Requirements Models

The delivered software should of course conform to the Requirements Model. We can't generally derive designs from requirements models because they tend to be partial, and so the designer has to apply skill and judgment to complete the design. But a test only needs to try exactly what is in the requirement: so ideally we should be able to derive tests from a requirements model. If it could be done automatically, this would create a solid verifiable link between the requirements and the code. If we can't do that, the model and the code might drift apart.

However, the requirements model (and hence the tests) is normally much more stable than the code. Adding a new feature may mean modifying existing code; but we only need to add new tests for the new feature, without altering existing tests. Tests need to be altered only when existing requirements change. This 'monotonic' property of the development process mitigates the problem of model/test divergence: the comparative rarity of alterations makes it acceptable to derive tests systematically by hand from the requirements; this is the role of the Test Team.

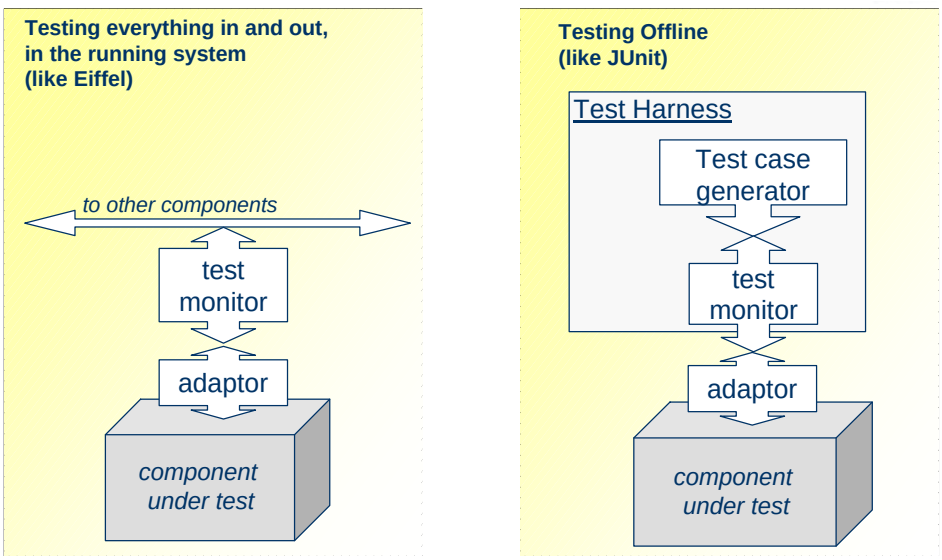
The rest of this section investigates the systematic derivation of tests from the requirements model.

Testing framework

We distinguish between a *Test Monitor* and a *Test Harness*. A Test Monitor sits at the interface to a component or system, monitoring everything that goes in and out and checking whether the component's responses meet the requirements. It is used within a running system. The Eiffel language provides for Test Monitors: the language allows for pre and postconditions to be attached to each message; a compiler option inserts code that checks them on entry and exit to each method. In debug mode, a system with activated Test Monitors runs slowly, but the functionality is checked.

A Test Harness is a set of software elements that surround a component under test; it runs offline, separately from the target context. It fires a sequence of messages at the component, and checks that all the right stuff comes back. It can check performance.

Using a Test Harness is the only serious way to be able to test a component automatically in a repeatable manner. In an incremental design context, repeatable tests are essential.



(The purpose of the Adaptor is to convert to and from whatever interface protocol is used at the surface of the component under test. One component might present a COM interface and another an HTTP interface; the Adaptor accepts commands from the Test Monitor to generate the appropriate COM or HTTP sequences; and translates outputs back again.)

Test case generation

A Test Harness can be thought of as a Test Monitor plus a prescription for the test messages that should be sent: a Test Case Generator.

Since we can't check all points in the space of possibilities, we have to have some strategy for choosing sufficient representative points to check all the zones and boundaries in the state space. This is perhaps the most difficult aspect to automate, and is a subject of ongoing research.

"White box tests" look at the program code being tested, and try to generate cases that exercise all routes through the code. But in a CBD context, we often don't have access to the code; and if the specification is of a pluggable interface, we don't know at the time of writing the tests, which component will be plugged in.

"Black box tests" are designed without knowing the internal structure of the component under test. In generating test cases, we look for boundaries in the problem space. For example, we would guess that any piece of software that can deal successfully with 3, 17, 33 and 127 items in a purchase order can probably deal just as well with any number in between, unless it is very peculiarly written; and we'll check specifically that it can deal with 0, 1, and 2 items.

Since we can't test every combination even of these sample points, we need also to look for likely dependencies intrinsic to the problem space. Independent variables needn't be tested in multiple combinations. For example, if the component deals correctly with a purchase order total from £0 up to £1000 with one or two items on the order, it will probably deal with any total for other numbers of items.

Pre- and Post-conditions

We favor writing pre/post-conditions for use-cases precisely because they state exactly what is to be achieved, rather than how. They can be translated straight to test monitors.

Each post-condition comes paired with a precondition; the latter states the range of applicability of the former. To take a simple example:

use-case	calculate total price of order
pre	count of items on order > 0
post	total price := sum of item.product.price

This tells us what is achieved in the case where the precondition is true.

What does it say about any cases where the precondition is not true?

Recall that when specifying requirements, it is valuable to be able to deal with partial views separately. This specification tells us what one class of user expects of the system. It is important that it does not prohibit other views. We might have another class of user role specifically interested in what happens when there are no items on the order. Or we may have several variants of the system, all of which conform to this fragment of specification, but which have different behaviors for the 0 case.

The proper interpretation is therefore that the precondition tells us when the postcondition is applicable. In cases where the precondition is false, we simply do not know (just by looking at this fragment) what to expect the system to do. Again, this is the 'monotonic' property so important for pluggable and extensible object or component design.

The tests that correspond to pre/post pairs are therefore of the form:

```
if (<precondition = true>)
{
    stored-values := <@pre-variables used in postcondition>;
    <invoke method under test>;
    if (<postcondition(new values, stored-values) = false>)
          { raise an exception; }
}
else { <invoke method under test>; }
```

(An "@pre-variable" is a variables referred to in the postcondition in the form "variable@pre", referring to its value before the method is invoked. [Warmer])

There are a number of related constructs; see [Catalysis] for more details:

- **Guards**. A guard looks like a precondition, but no changes should happen if it is false.
- **Model templates**. A template is a chunk of model that can be parameterized and included in a model (like a macro). Composing models with templates is not monotonic: preconditions can be strengthened.

Attributes and Associations

Postconditions etc are, as we've seen, written in terms of the attributes and associations of the object model. For example "The Guest is now the occupant of a room in the Hotel, that was previously unoccupied" specifies a change in the *occupant* relation.

To be able to test this requirement, we therefore need to be able to access the *occupant* relation. But wait a minute: requirements specifications say nothing about design – so there might not exist any such relation, at least in ordinary database terms. The associations and attributes of a Requirements Model tell us only about what information is kept within the system in some form. Looking at the type diagram, it says that we should be able to ask the system what *accommodation* a particular *Hotel* has, and the answer should be a collection of *Rooms*; and we should be able to ask about the *occupant* of each Room, and the answer should be either a *Person* or *Null*.

So any design conforming to this specification should provide query operations corresponding to the associations and attributes – if only so that we can test it.

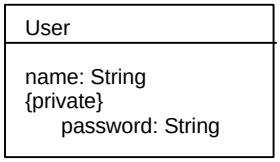
The test of the usecase *check guest into hotel* according to this postcondition would be something like this (where *hotelsys* is the component under test):

```
for all room in hotelsys.accommodation(hotel)
    previousOccupant[room] := hotelsys.occupant(room);
hotelsys.checkInGuest (guest, hotel);
if (not previousOccupant[hotelsys.roomWithOccupant(guest)] = null)
    raise exception;
```

Notice that we are asking the component *hotelsys* to perform each query and operation: we are not assuming anything about operations the hotel, guest, etc objects can perform.

Private attributes

Not all attributes can be exposed. For example, we might model a *User* as having an attribute *password:String*. The login use-case is specified as



associating a Terminal with a User if the word typed at the terminal matches that in the User's *password* attribute.

The design should definitely not have a way of exposing the password at the system interface. So we could not write the test to include something like:

```
system.login(user, myWord, terminal);
if (myWord = system.getPassword(user)
    and not system.isLoggedIn(user, terminal)) raise exception;
```

To indicate the restriction, we can mark the attribute *private*, prohibiting a design from implementing *getPassword*. Again, recall that we have said nothing about how the attribute itself should be implemented.

To test use-cases defined using a private attribute, we have to perform sequences of operations that demonstrate behavior according to the specification. For example:

```
system.setpassword (user, "thing");
system.logout (user);
system.login (user, "fiddle", terminal);
if (system.isLoggedIn(user)) raise exception;
system.login (user, "thing", terminal);
if (not system.isLoggedIn(user, terminal)) raise exception;
```

This kind of test is difficult to derive automatically from the Requirements Model.

Invariants

A typical invariant might be "The guests of a Hotel are the occupants of its accommodation", or more succinctly,

```
Hotel::   guests = accommodation.occupant
```

This is simple enough to translate to a test. It should be tested after any use-case that might affect any of the associations mentioned.

(The rules are more complicated where there is callback between components.)

Nonfunctional (Quality) Requirements

In general, NFRs such as Loading, Latency, Usability, Security need specialist testing apparatus and test procedures.

In all of these cases, UML has little to say about how to specify the requirement. The best way of specifying it is in the form of test parameters to the relevant tools. However, the basic vocabulary provided by the use case and type models can be used: for example, "*check in guest* should require less than 5 keystrokes plus the reservation number or guest name".

Iterative Design from Requirements Models

It's essential that requirements are developed as a successive approximation, and the requirements are passed to the developers very regularly. The division between analysts and designers is about skills, not about sitting in separate rooms.

As in XP, designers should choose high-risk spikes to work on early, and should do all the good XP stuff – Simplest Thing That Could Possibly Work, Write Tests First, One Change at a Time, each change a Refactoring (no change to tests) or an Elaboration (extensions to tests, previous tests continue to work).

Design tasks

(We don't distinguish between programming and design here.)

The XP Planning Game is used to schedule design tasks [Beck]. Customers assign priorities to features of the requirements; designers estimate how long they will take; together, they work out a schedule. The schedule is reviewed every two weeks, to review progress and adjust it in the light of changed understanding of requirements, architecture, and task estimates.

In the XPRM version,

- Analysts represent the customers: either entirely, if the customer is diffuse; or sitting with the customer in the style of an attorney at a police interview.
- The customer prioritizes *Features.* A Feature is some characteristic of the delivered system that can be demonstrated to the customer, and tested by the Test Team. A feature can be:
  - A use case. E.g. user can see list of products.
  - A group of related use-cases. E.g. user can logon and logoff.
  - A clause in a postcondition, or related set of such clauses. E.g. Each login is recorded, and the manager can get a report of all logins.
  - A business rule. E.g. no user is logged in from two terminals concurrently.
  - A NFR/quality requirement. E.g. 1000 concurrent users can get decent performance.

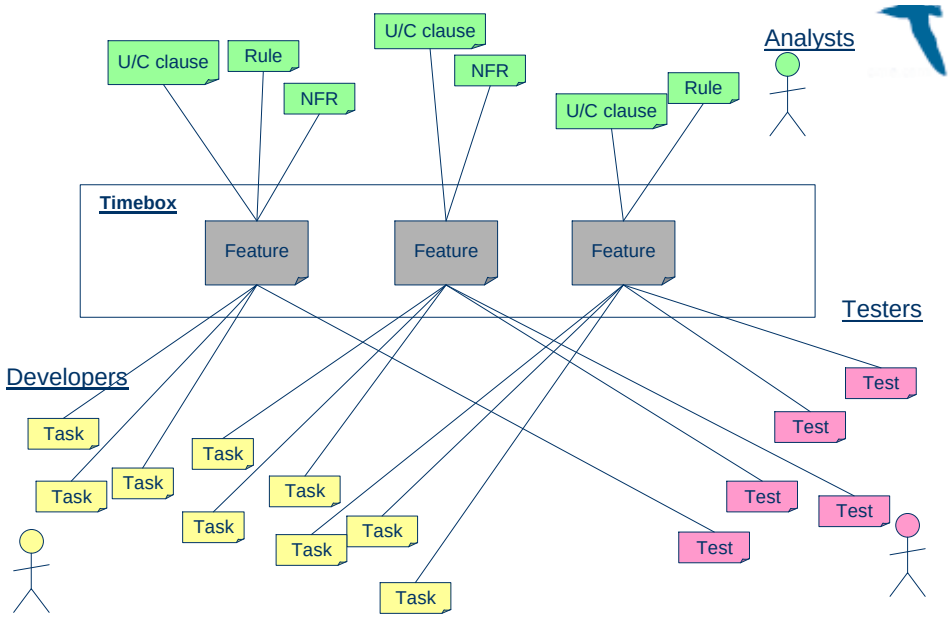
  Each Feature is documented in the project plan by reference to the parts of the Requirements Model that it demands should be implemented.
- The designers define and estimate design *Tasks.* A Task is a unit of design or programming work led by one designer, taking less than 5 designer-days. Design might include actual programming, or it might be high-level design, or experimentation with existing systems. Each task has a *demonstrable goal* though not necessarily demonstrable to the customer. Each Feature has a principal task directed at fulfilling it; but there are ancillary and architectural tasks too.

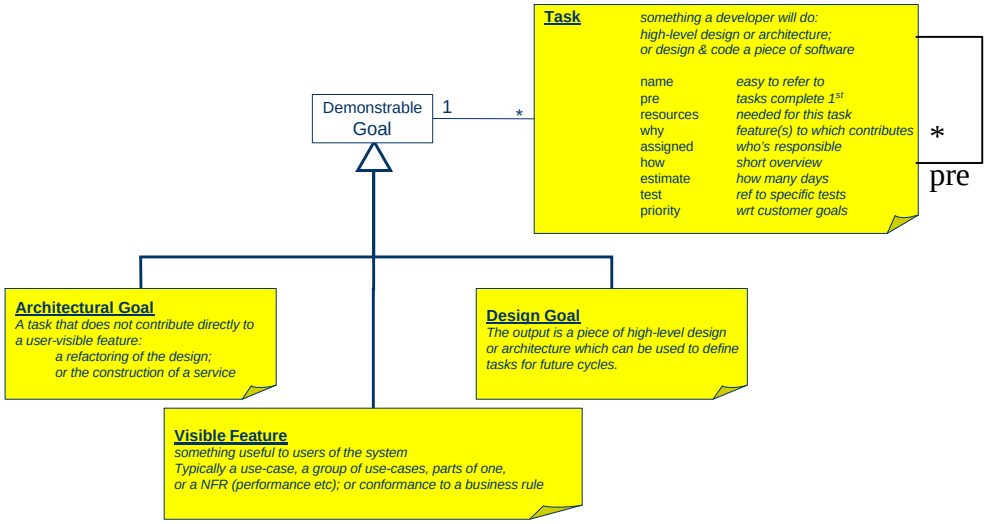
  A typical Feature might require some user interface design, an extra piece in the database, some content, some business logic, and those efforts will have to be integrated. Each of these is a task (probably because the different skills reside in different heads).

  Tasks have *prerequisites:* other tasks that must be complete first. When planning, tasks may be rearranged, but only insofar as the prerequisite graph allows.

  Some tasks, called “architectural tasks” are prerequisite to a wide range of others, without being attached to any Feature. These are typically about refactoring and infrastructure.
- The Test Team is also part of the planning game. They check that Features are well-defined enough to be tested, and contribute testing tasks to the schedule: a Feature isn’t done until tested. (Testers test features, rather than the results of tasks: unit tests are written by developers.)

As in XP, the performance of the schedule is constantly monitored.





Costing

Analysts should try to get a broad even degree of coverage of the whole range of the system within the first few weeks of the project (in the RUP 'elaboration' stage). During this period, the designers should choose an area to prototype. Using the usual XP techniques of estimating and monitoring programming tasks, the team's velocity is calibrated.

From the size of the prototype area in the Requirements Model, and time taken to implement it, an estimation of the time to deliver other areas of the requirements can be obtained. Various spreadsheets and tools like Optimise can help in 'sizing' the model.

This is very useful where fixed-price contracts are demanded.

Tool support

XPRM would benefit from tool support in these areas:

- An analysis tool that helps apply our model elaboration patterns and checks.
- Assistance in generating test cases from requirements models; and in tracing changes to the models through to the test cases. Integrates results from a range of testing tools.
- A project management tool that supports the planning game with Features and Tasks; can link the Feature definitions to parts of the Requirements Model; supports the rearrangement of Tasks in a schedule; supports inspection and assignment of tasks over the project intranet.

Summary

eCommerce requires the rapid development of large systems. A lightweight process such as eXtreme Programming (XP) is an essential for success. XP is a rigorous approach, mandating frequent regression testing. However, it assumes small projects in which a single set of people perform both analysis of the customer's requirements and the development of the solution. This is impractical in many software shops, if only because a skilled analyst and a skilled developer don't always co-exist in the same body.

We've looked at:

- integrating UML-based requirements without jeapordising XP's responsiveness to change;
- strong consistency and completeness checks on requirements models;
- the relationship between use-cases and incremental development tasks;
- maintaining a rigorous relationship between tests and the requirements model.

References

[Beck] *eXtreme Programming eXplained*, Kent Beck

[Catalysis] *Objects, Components and Frameworks in UML*, D D'Souza & AC Wills [A-W 98]

[Cockburn] *Writing Effective Use Cases,* Alistair Cockburn [A-W 2001]

[Jones] *Systematic Software Development with VDM,* Cliff Jones [PHI 1986]

[Kruchten] *Rational Unified Process* P Kruchten [A-W 2000]

[Morgan] *Programming from Specifications,* Carroll Morgan [PHI 1990]

[Warmer] *Object Constraint Language,* Jos Warmer and Anneke Kleppe [A-W 1998]