SOFTWARE DEVELOPMENT PLAN

RadMap

Release v3.0.0



radmap.tech

Contents

1 Scope 2

2 Organisation 3
2.1 Competence . . . 3

3 AI policy 3

4 Development process 3

5 Verification and validation 4

6 Hazard analysis 5

7 Configuration management 5

8 Third-party components 6
8.1 Component register . . . 6
8.2 Upgrading a component . . . 7

9 Controlled documents 7
9.1 Register of procedures . . . 8
9.2 Records . . . 8

10 Change impact 8

11 Defects 9
11.1 Which versions are checked . . . 10
11.2 Notifying users . . . 10

RadMap's software development plan: how RadMap is developed, verified, released and corrected. It is written for anyone who has to judge whether RadMap is fit for their purpose – the EULA's *Intended Use* leaves that judgement to the user, and a nuclear licensee qualifying RadMap will read it for that. It describes how RadMap is actually made, including what it does not do.

1 Scope

RadMap calculates dose rates from radioactive sources and the effect of shielding on them: a Windows desktop application (`RadMap.exe`) and the Geant4-based simulation it runs (`DoseAlgorithm.exe`), with their installer and user guide. The EULA states the intended use; the user guide's *Known limitations* states what RadMap does not calculate, or calculates only approximately.

This plan is RadMap's quality programme: the description a user qualifying RadMap for their own purpose can base that qualification on. It does not claim compliance with a nuclear quality assurance standard such as ASME NQA-1 or ISO 19443.

The plan is published with each version's documentation, as it stood at that version's release. The procedures it names are documents in RadMap's source repository, which is not public.

2 Organisation

The RadMap team develops RadMap and is responsible for every activity in this plan: requirements, design, implementation, verification, release, and handling defects. Development is led by Klas Marcks von Würtemberg, the Licensor named in the EULA.

2.1 Competence

Decisions on RadMap's physics are made by a team member with:

- a doctorate in nuclear or radiation physics, or equivalent research experience;
- working knowledge of Monte Carlo radiation transport, and of the quantities and data of radiation protection dosimetry (ICRP, ICRU and evaluated decay data).

That covers the choice of transport physics and data, and the derivation and acceptance criterion of each V&V case. A change that can change a result (*Change impact*) is reviewed by a member with this competence before it is committed.

3 AI policy

The team may use AI coding assistants to draft code, tests and documentation.

- **Everything is reviewed.** Whatever an assistant drafts is reviewed by the team before it is committed, and the team is responsible for it as for any other change.
- **An assistant takes no decisions.** It does not commit, merge, tag or release: every change enters the repository, and every release is made, by the team.
- **The same procedures apply.** An assistant works to the procedures in the register below, including the change impact assessment every commit records.
- **Development machines only.** An assistant builds and tests its drafts on the team's development machines. It has no part in the build machine: releases are built and tested there by CI, from the committed source alone.

4 Development process

Work is done on feature branches, merged into `develop`, and released by a pull request into `master` that is then tagged (VERSIONING.md, *Branches and releases*). `master` changes only through pull requests and always names the latest release. Before a change is committed it passes the build and the tests that cover it, and its effect on results is assessed (*Change impact* below).

5 Verification and validation

- **Tests.** Unit tests check modules in isolation; integration tests check the application, the simulation and the files between them. The V&V cases are the test cases marked as such in the tests `docs/vv_report/sources.json` lists: the physics tests (`DoseTest`), which check dose rates against references derived from published conversion coefficients, attenuation data, buildup factors and decay data, and the tests of geometry, materials, source sampling, activity, reproducibility and validation. Each case has its derivation and acceptance criterion written into it (TESTING.md, *V&V cases*), and states its basis: a published reference throughout, partly RadMap's own earlier result (the buildup behind a finite water slab, a geometry no published factor covers), a condition rather than a value, or a published measurement.
- **When they run.**
  - Every push: the unit and integration tests, on the release and debug builds.
  - Every push to `develop` and `master`, every pull request into `master` and every release tag: also the physics tests (those judged on simulated dose rates), which take minutes, on the release build.
  - `master`, pull requests into it and release tags only: also the validation tests (`ValidationTest`, against published measurements), which take longest; `develop`'s rehearsal report shows their cases as not run.

  A release is refused unless every V&V case ran and passed (VERSIONING.md, *Releasing*).
- **Records.** Each release keeps its test reports, with every measured value and the statistical uncertainty of each simulated one, as its V&V record (*Records* below).
- **Report.** Each release's V&V report is generated from those results in the same CI run (`tools/vv_report.py`) and published with its documentation; it takes only results recorded by the build it describes, which every check identifies by its commit and any uncommitted change (`tools/vv_identity.py`). It lists the requirements (`docs/vv_report/requirements.md`) and the safeguard requirements (`docs/vv_report/safeguards.md`), which cases verify each, and any that none verifies yet; and the hazard analysis (`docs/vv_report/hazards.md`): each way a user could act on a wrong dose rate unaware, the requirements that control it, and what is open, with the defects it cites as they stood at the release. Pushes to `develop` and `master` build the report and the record too, as a rehearsal kept only as a workflow artifact, so a change that breaks them shows before a release. A report built anywhere but a release tag's run says on its title page and in its summary that it is a draft, not a release.
- **Validation**: most cases compare RadMap with reference models, most of them calculated, which is verification. `ValidationTest` compares it with published measurements in air kerma, the measurable quantity RadMap computes (basis `measurement`): the two Kansas State radionuclide experiments, which is as many photon experiments as MCNP's shielding validation suite has. The skyshine case also compares RadMap with LLNL's COG on the same model. Geant4's published validation is open work (the V&V report's *Verification and validation* says so).
- **User interface and report.** No automated case covers the windows a user types into and reads from, or the PDF report. Every change states whether it can alter what a user sees; one that can is checked by hand when it is made, by the developer who made it, for what it touches, and the commit message records what was checked (*Change impact*). The V&V report lists every such change since the previous release with its record, and a release is refused while a change lacks its answer. The first release under this rule is also checked by hand as a whole, once, from the list in TESTING.md (*Checks by hand*), since the changes

before it carry no records. The libraries behind the windows and the report are evaluated in the component register (*Third-party components*); the installation verification is each user's own check of the whole chain, from the computed dose rate to the window and the report.

6 Hazard analysis

The V&V cases show that RadMap computes the right dose rate for a project. The hazard analysis asks the other question: how could a user act on a wrong dose rate without knowing it is wrong? It lists each such path as a **hazard** (`docs/vv_report/hazards.md`): a fault in RadMap, a damaged installation, a result misread, a calculation set up for another question than the user's. For each it names the requirements that control it – those on the computation, and the **safeguard requirements** (`docs/vv_report/safeguards.md`) that keep the project computed and the number read the user's – the other evidence, the defects that came by it, and what is still open. The V&V report publishes it with each release, each control shown with how its cases went in that release's build.

The analysis is kept current at three points:

- **A defect.** Every defect that can affect a result is named under the hazard it came by (*Defects*, step 5). If no hazard describes its path, one is added. If the hazard's controls should have caught it, the gap becomes a V&V case, or an open item of the hazard until one exists. `VvSourcesCheck` fails while a defect that can affect a result is named under no hazard.
- **A change.** A change that opens a new way to a result – a new kind of input, a file, a front end – or that removes or weakens a control, updates the hazards and safeguard requirements in the same commit. A safeguard requirement is worded no more strongly than its cases show; when writing a case finds the code guarantees less, the code or the wording is corrected first.
- **A release.** A V&V case counts in a release's report only if the release ships what it exercises (TESTING.md, *V&V cases*), and the report lists every safeguard requirement no released case verifies. Before tagging, the hazards are read against the changes since the previous release.

The hazards' text is written for the report's reader: it names the release notes, the errata page and the user guide, never a file of the source repository. A standard is cited only by its edition and clause, for what it requires.

7 Configuration management

- Source, procedures and documentation are in git; a release is a tag on `master`.
- Every build input – compiler, libraries, Geant4 and its datasets, Python and documentation tools – is pinned by version and, where downloaded, by SHA-256 (BUILDING.md), so a release can be rebuilt from its tag.
- Versions of the application, the simulation and the file formats are numbered as VERSIONING.md describes.
- Each simulation run gets a run ID, stored with its results and printed on its report with the RadMap version, which ties a report to the project and version that produced it.

8 Third-party components

Geant4 performs the radiation transport. RadMap builds it from source, from a fork carrying RadMap's patches, at one commit of it pinned in the build (BUILDING.md); the fork is the only place the patches live. Geant4's published validation applies to stock Geant4; RadMap's patches are RadMap's responsibility and are covered only by RadMap's own tests. The Geant4 datasets RadMap ships are pinned with the build. The other components are pinned the same way; the register below lists them all.

8.1 Component register

Every component RadMap's build, program, installer or documentation depends on, at the version this release is built with, and the class that decides how it is evaluated. `SupplierRegisterCheck`, run with every build's tests, fails when a version here differs from the pin the build uses.

Component	Version	Pinned in	Class
Geant4	11.4.0, RadMap fork commit b25df82	`conanfile.py`, Geant4 recipe	1
Geant4 data sets	G4EMLOW8.8, G4ENSDFSTATE3.0, PhotonEvaporation6.1.2, RadioactiveDecay6.1.2	with Geant4	1
Xerces-C++	3.3.0	`conanfile.py`	2
pugixml	1.16	vendored, `src/pugixml`	2
miniz	3.1.1	`conanfile.py`	2
MSVC compiler	VS Build Tools 18.4.2, compiler.version 195	`setup.ps1`, Conan profile	2
wxWidgets	3.3.2	`conanfile.py`	2
wxPdfDocument	1.3.1	`conanfile.py`	3
GLM	1.0.1	`conanfile.py`	3
glad	0.1.36	generated, `src/radmap/glad`	3
Fonts	Inter 4.1, Space Grotesk 2.0.0, JetBrains Mono 2.304	`setup.ps1`	3
Catch2	3.13.0	`conanfile.py`	4
Python	3.13.16	`setup.ps1`; packages `requirements.txt`	4
.NET SDK	10.0.403	`setup.ps1`, `global.json`	4
TeX Live	2026, snapshot 2026/10/12	`setup.ps1`	4
7-Zip	24.09	`conanfile.py`	5
Inno Setup	6.7.1	`setup.ps1`	5

The licences of the components that ship, and what they require, are in `resources/ThirdPartyNotices.txt` (BUILDING.md, *Third-party dependencies and attribution*).

Class 1: computes results. Evaluated on the supplier's evidence and on RadMap's own. The Geant4 Collaboration develops and tests Geant4 and publishes validation results for its releases (geant-val.cern.ch); that is evidence about Geant4, for stock Geant4, and not an endorsement of

RadMap. The data sets are evaluated nuclear and atomic data – ENSDF and the Livermore libraries – distributed by the Collaboration with the toolkit. What RadMap runs, its patches included, is verified by the V&V cases and validated by the validation cases on every release build (*Verification and validation*); the decay data and the photon interaction data are checked against published references by the cases for R-1 to R-3, R-8, R-10 and R-11.

Class 2: carries inputs, geometry or results. A fault would change what Geant4 receives or what is stored, so RadMap's own tests cover the path each one is on. Xerces-C++ parses the geometry Geant4 runs, which the geometry cases (R-12) verify; pugixml and miniz write and read projects and their results, which the project tests check; the compiler builds every binary the V&V cases run on. wxWidgets carries what the user types into the model: it is covered by the checks by hand (*Verification and validation*, *User interface and report*) and by its own test suite, which its project runs on every change. wxWidgets also draws what the user reads, class 3; a component with two roles takes the stricter class.

Class 3: presents results. A fault would show the user a wrong picture or a wrong number, though the result itself is right: wxPdfDocument draws the report, GLM and glad the 3D view, and the fonts set the text. Covered by the checks by hand and by the installation verification, which compares the numbers in the window and in the report with the expected values. wxPdfDocument has sample programs but no test suite of its own, so for it RadMap's checks by hand are the evidence.

Class 4: produces evidence. A fault would misstate the V&V record rather than a result: Catch2 judges each check, and Python, Sphinx and TeX Live write the V&V report and the documentation. The .NET SDK builds the Avalonia front end, which does not ship, and the test that checks its declarations of the C API. `tools/vv_report.py` refuses results that do not fit the cases (*Verification and validation*, *Report*), and the published report is read before a release.

Class 5: installs. 7-Zip unpacks the data sets and Inno Setup builds the installer; a fault would leave a damaged installation, which the installation verification finds.

8.2 Upgrading a component

1. For a class 1, 2 or 3 component, read the new version's release notes and known problems for the parts RadMap uses.
2. Change the pin (BUILDING.md) and the register above together; `SupplierRegisterCheck` fails until they agree.
3. Update `resources/ThirdPartyNotices.txt` when the component's notices or licence changed, or it brings a library it did not before.
4. For a class 1 or 2 component, run every V&V test, the validation cases included; if `VerificationTest` then fails because results moved as intended, write the new verification reference (TESTING.md, *Installation verification*).
5. For a class 2 or 3 component that draws or carries what the user sees – wxWidgets, wxPdfDocument, GLM, glad, the fonts – check the windows and the report by hand, as for a change to them (*Change impact*).
6. Say in the commit's change impact line what ran and what moved, and the check by hand (*Change impact*).

9 Controlled documents

RadMap's procedures are Markdown files in its source repository, and git is their document control:

- Every change to a procedure is a commit, with its author, date and reason. The team approves a change by committing it; there is no separate sign-off.
- The procedures in force for a release are those at its tag, so how a version was made can be read back exactly.
- A procedure lives in one place. Where a topic's document already holds it, the register below points there rather than repeating it.

9.1 Register of procedures

Procedure	Where
Change impact assessment	this plan, *Change impact*
Defect handling	this plan, *Defects*
Hazard analysis and safeguard requirements	this plan, *Hazard analysis*
Design rules and type placement	ARCHITECTURE.md, CONVENTIONS.md
Testing	TESTING.md
Dosimetric data: adding and checking tables	DOSIMETRY.md, *Before adding a catalogue row*
Building, pinned inputs, installer, CI machine	BUILDING.md
Adding a third-party dependency	BUILDING.md, *Third-party dependencies and attribution*
Upgrading a third-party component	this plan, *Upgrading a component*
Versions, branches and releasing	VERSIONING.md
User-facing record of changes	CHANGELOG.md, *How to write entries*

9.2 Records

Record	Where
Change impact of each change	its commit message
Defects in released versions	DEFECTS.md
The same, as users see them	the errata page (*Notifying users*)
What changed for users, per release	CHANGELOG.md
V&V results of each release	release vX.Y.Z of the source repository: vv-record-vX.Y.Z.zip
V&V report of each release	its documentation site: RadMap_VV_report_vX.Y.Z.pdf
Hazard analysis of each release	its V&V report, *Hazard analysis*

10 Change impact

Every change answers one question before it is committed: **can it change a result RadMap computes?** A result is anything a user reads off a run or relies on for one – a dose rate, the geometry, materials or sources Geant4 receives, an activity, a number in the report. When in doubt, the answer is yes.

The commit message records the answer in one line:

- `Results: unchanged` – and nothing in the change can reach a result.
- `Results: <what can change>; checked by <how>` – naming the tests that cover it, or how it was checked when no test does. A change meant to alter results says by how much, for which cases.

A change that alters results and is not a defect fix still gets its CHANGELOG entry saying so, because a user re-running an old project will see different numbers.

Every change answers a second question as well: **can it change what a user sees?** – a window, a dialog, the 3D view, the PDF report or the text in any of them, whichever file the change is in: a change in the model can alter a column of the report. The commit message records the answer in a second line:

- `UI: unchanged` – and nothing in the change can reach what a user sees.
- `UI: checked by hand: <what>` – the dialogs, panels or report pages the change touches, exercised in the built application, the report opened and read.

A line forgotten or wrong is given by a later commit, with a line `Results (for <commit>): <answer>` or `UI (for <commit>): <answer>`, which replaces the commit's own; a commit message is never rewritten. `CommitRecordsCheck`, run with every build's tests, fails while a commit since the rule's introduction (2e95023) or the previous release has no `UI:` answer, or a placeholder such as `<what>` for one, and the V&V report lists the changes that were checked by hand. A merge commit needs neither line, since every change it brings carries its own. A merge whose conflicts were resolved with code neither side had is a change of its own, and its message gives both lines; `CommitRecordsCheck` passes over merge commits, so nothing checks that it does.

11 Defects

A **defect** is a fault in a released version: one that has reached users. A fault found and fixed before release is handled as an ordinary change and is not logged.

A fault is anything RadMap does other than it should – a wrong result, a message that misdescribes what happened, a window that shows other than the project holds, a process left running – except a purely cosmetic flaw, such as untidy drawing or uneven spacing: it tells the user nothing false and changes nothing they do, and is fixed as an ordinary change, with a changelog entry if users would notice it.

When a defect is found:

1. Add it to DEFECTS.md at once, before it is fixed. An entry is never deleted; it is completed as the defect is understood and fixed.
2. Assess its effect: can it change results, under which conditions, by roughly how much, and in which versions.
3. If it can change results, tell users (see *Notifying users*), and record that they were told.
4. Fix it under change impact assessment, and note the fixing version in the entry.
5. If it can change results, name it under the hazard it came by, and ask why that hazard's controls missed it (*Hazard analysis*).

A bug noted in TODO.md that affects a released version is a defect too: it gets a DEFECTS.md entry, which outlives the TODO entry.

11.1 Which versions are checked

The released versions are 1.1.0 and every version since; those before 1.1.0 were pre-releases, and are not listed. A version is **supported** for two years from its release, and the latest release is supported however old it is. Support is counted in time, not in version numbers: the MAJOR number counts breaking changes to the API (VERSIONING.md), not how long a version stays in use.

Whether a defect is in a version is checked in that version's source, at its tag:

- A defect that can change results – an effect other than "none", "negligible" included – is checked in every released version. Reports made with an old version are kept, and the errata page is how their readers learn that one needs review.
- Any other defect is checked in the versions supported when it is found. Its entry says so, `2.0.0, older not checked`, and the errata page marks the older versions with a ?, not a blank: a blank says the version does not have the fault.

`tools/build_errata.py` refuses "older not checked" on a defect that can change results, and one that leaves a supported version unchecked.

The errata page's matrix has a column for each supported version, and one, *Older*, for the rest. It keeps a row for every defect, however old: a reader with a report from an old version looks down *Older*, and the entry names the versions.

11.2 Notifying users

Users learn of defects from the **errata page**, `https://radmap.tech/documentation/errata.html`, which is built from DEFECTS.md (`tools/build_errata.py`) on every documentation deploy. It is not part of any version's guide, because a guide is frozen at its release while defects in it are found later; every version's footer links to it.

- A defect is published by merging its DEFECTS.md entry to `master`, as a doc fix (BUILDING.md, *The errata page*), as soon as its effect on results has been assessed – not when the fixing version ships. Defects found before the first published release, 2.0.0, are published with it: until its tag exists, a push to `master` publishes nothing (BUILDING.md, *Publishing HTML documentation to GitHub Pages*).
- The fixing version's CHANGELOG entry, under *Fixed*, also announces a defect that can change results, starting **Affects results:**, with the conditions under which it applied and the versions affected.
- A PDF report prints the RadMap version that made it and its run ID, which is how a user tells whether a result needs review.

Licensees are not told individually: the errata page is the one place to look.